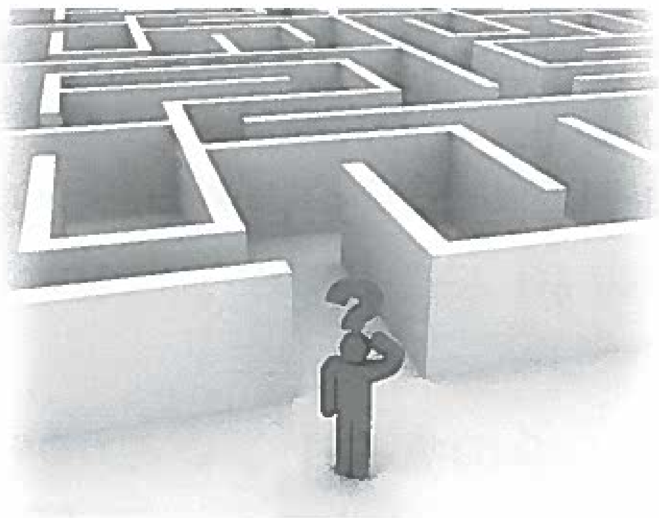
Week 2

July 17, 18 13



Divide and Conquer

Counting Inversions I

Design and Analysis of Algorithms I

InvI-1

Divide and conquer paradigm
① divide into smaller subprob.
② conquer via recursive calls.
③ combine sol of subproblem into the original problem } the most ingenuity.

The Problem

<u>Input</u> : array A containing the numbers 1,2,3,..,n in some arbitrary order

Def. of inversion

<u>Output</u> : number of inversions = number of pairs (i,j) of array indices with i<j and A[i] > A[j]

ex. If A is sorted # inversion = 0

Tim Roughgarden

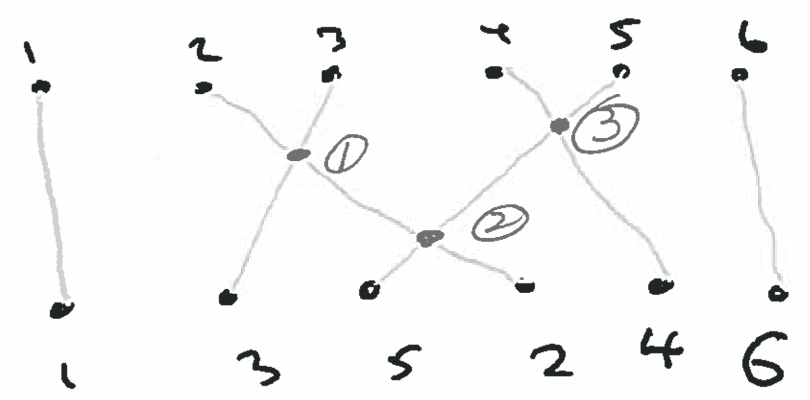
To find the # of inversions, we write the numbers in sorted order once and the second time we write them as they given. Next, we connect numbers. The # of crossing lines = # inversions

Examples and Motivation

Example (1, 3, 5, 2, 4, 6)

Inversions :

(3,2), (5,2), (5,4)



Motivation : numerical similarity measure between two ranked lists eg: for collaborative filtering

See section 3.2.3 for the def. of (to suggest items/movie to buy).

Tim Roughgarden

INVI-2

The worst case is that the array is in the backward order

$A(1) \cdots \quad A(n)$

$n \quad (n-1) \quad (n-2) \cdots 1$

$$
\begin{array}{lll}
1 & & (n-1) \\
2 & + & (n-2) \\
3 & + & \vdots \\
n-1 & + & (n-(n-1)) = 1
\end{array}
\quad = \frac{n(n-1)}{2}
$$

What is the largest-possible number of inversions that a 6-element array can have?

- ○ 15
- ○ 21
- ○ 36
- ○ 64

In general, $\binom{n}{2} = n(n-1)/2$

Te
ve

Two loops.

High-Level Approach

Brute-force : $\theta(n^2)$ time = Asymptotic running time.

Can we do better ? Yes!

KEY IDEA # 1 : Divide + Conquer

classify the inversions of an array A of length n into one of three types:

Call an inversion (i,j) [with i<j]

- Left : if i,j ≤ n/2
- Right : if i,j > n/2
- Split : if i<=n/2 < j

Note : can compute these recursively

need separate subroutine for these

|| combine step of algo

Tim Roughgarden

High-Level Algorithm

```
Count (array A, length n)
        if n=1, return 0        (base)
        else
                X = Count (1st half of A, n/2)
                Y = Count (2nd half of A, n/2_
                Z = CountSplitInv(A,n) ← CURRENTLY UNIMPLEMENTED
        return   x+y+z
```

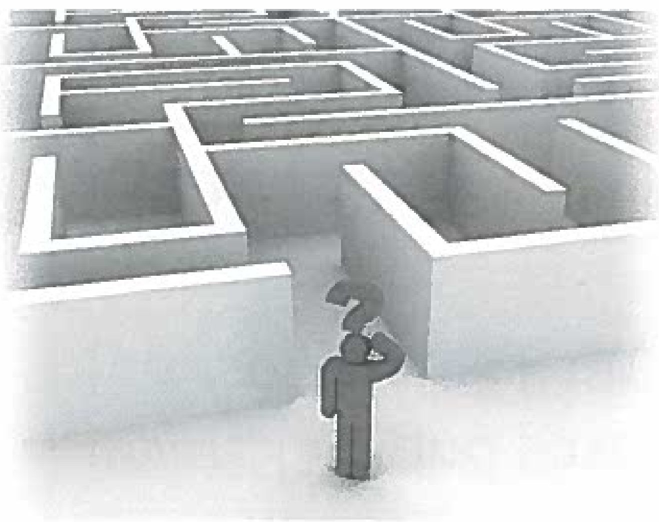
Goal : implement CountSplitInv in linear (O(n)) time then count will run in O(nlog(n)) time [just like Merge Sort]

Tim Roughgarden

July 17, 18 14

Divide and Conquer

Counting Inversions II



Design and Analysis of Algorithms I

inv II 1

Piggybacking on Merge Sort

KEY IDEA # 2 : have recursive calls both count inversions and sort.

[i.e. , piggy back on Merge Sort]

Motivation : Merge subroutine naturally uncovers split inversions [as we'll see]

great

Tim Roughgarden

/n

High-Level Algorithm (revised)

Sort-and-Count (array A, length n)

if n=1, return 0

else

Sorted version of 1st half — (B,X) = Sort-and-Count(1st half of A, n/2)

Sorted version of 2nd half — (C,Y) = Sort-and-Count(2nd half of A, n/2)

Sorted version of A — (D,Z) = CountSplitInv(A,n) +Merge ← CURRENTLY UNIMPLEMENTED

return X+Y+Z

merge-and-

<u>Goal</u> : implement CountSplitInv in linear (O(n)) time

=> then Count will run in O(nlog(n)) time [just like Merge Sort]

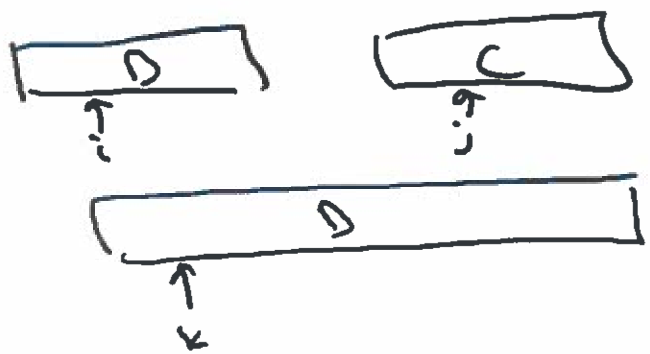
Sort-and-

Tim Roughgarden

Inv II_2

Pseudocode for Merge:

D = output [length = n]
B = 1st sorted array [n/2]
C = 2nd sorted array [n/2]
i = 1
j = 1



```
for k = 1 to n
        if B(i) < C(j)
                D(k) = B(i)
                i++
        else [C(j) < B(i)]
                D(k) = C(j)
                j++
end
```

(ignores end cases)

Tim Roughgarden

Ter
ver

Suppose the input array A has no split inversions. What is the relationship between the sorted subarrays B and C?

- ◯ B has the smallest element of A, C the second-smallest, B, the third-smallest, and so on.
- ◯ All elements of B are less than all elements of C. ✓
- ◯ All elements of B are greater than all elements of C.
- ◯ There is not enough information to answer this question.

InV II–3

Example

already sorted

Consider merging B $\boxed{1 \mid 3 \mid 5}$ and C $\boxed{2 \mid 4 \mid 6}$.

Output : $\boxed{1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6}$ D

⇒When 2 copied to output, discover the split inversions (3,2) and (5,2)

⇒ when 4 copied to output, discover (5,4)

After copying 2 i is greater than j

General Claim

<u>Claim</u> the split inversions involving an element y of the 2nd array C are precisely the numbers left in the 1st array B when y is copied to the output D.

<u>Proof</u> : Let x be an element of the 1st array B.

√1. if x copied to output D before y, then x < y

⇒ no inversions involving x and y

2. If y copied to output D before x, then y < x

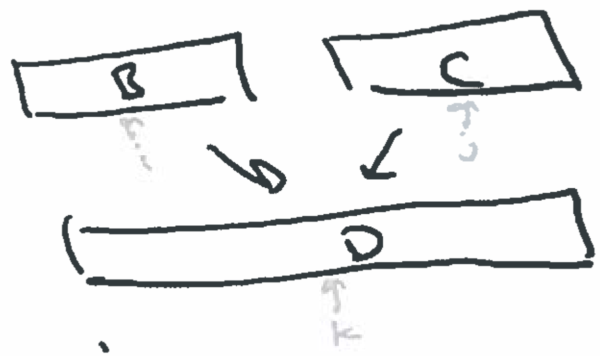
=> ~~X~~ x and y are a (split) inversion. Q.E.D

Tim Roughgarden

InV II - 4

Merge_and_CountSplitInv

-- while merging the two sorted subarrays, keep running total of number of split inversions



-- when element of 2nd array C gets copied to output D, increment total by number of elements remaining in 1st array B

Run time of subroutine : $\underbrace{O(n)}_{\text{merge}} + \underbrace{O(n)}_{\text{running total}} = O(n)$

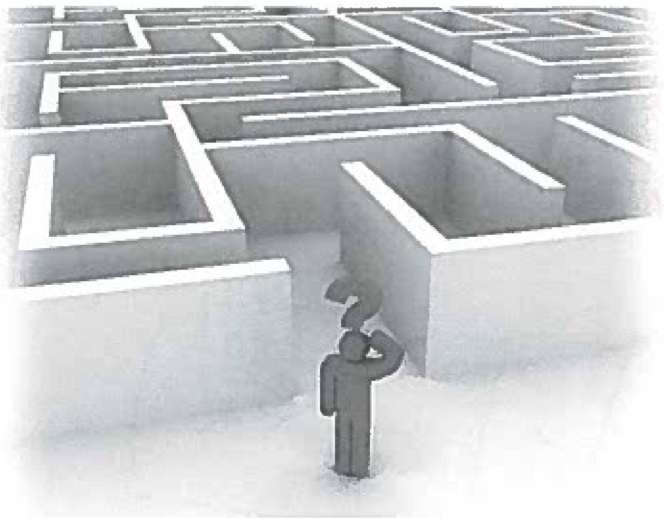
=> Sort_and_Count runs in O(nlog(n)) time [just like Merge Sort]

Tim Roughgarden

If you add O(n) nX then the total cost is O(n²)
but if you add it constant times then it is O(n)

Strassen's algo.
- not trivial
- fundamental prob.

July 18, 18 15



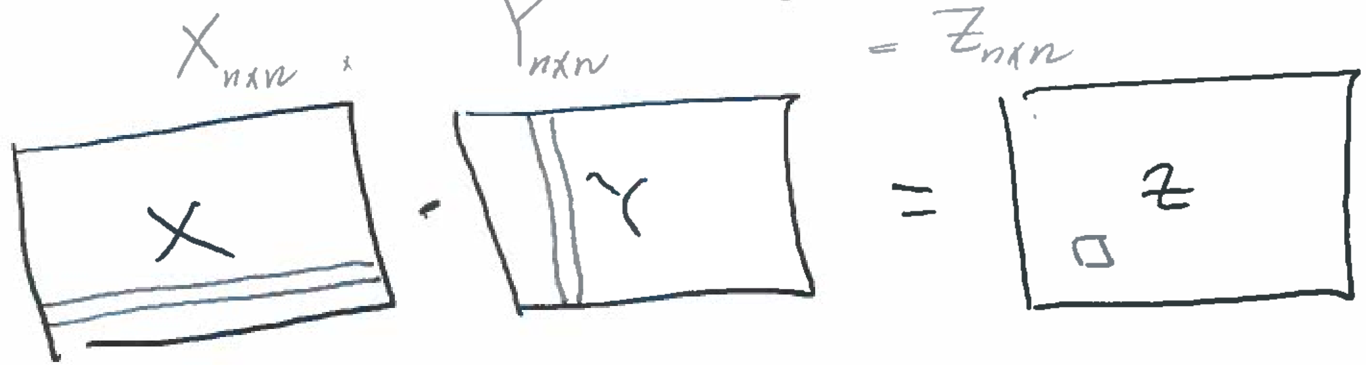
Design and Analysis
of Algorithms I

Divide and Conquer

Matrix Multiplication

matriz-1

Matrix Multiplication



some dim (all n X n matrices)

Where z_{ij} = (ith row of X) . (jth column of Y) ← dot product.

$$= \sum_{k=1}^{n} X_{ik} \cdot Y_{kj}$$ ← dot product.

Note : input size $= \theta(n^2)$ ← not the running time

Tim Roughgarden

- running time: for each entry in Z: $\underset{\text{mult}}{n} + \underset{\text{sum}}{n} = 2n$
- Total running time: (n^2 entries) $* n = \Theta(n^3)$.

The best we can hope, bcs there are n^2 entries in the matrix is $\Theta(n^2)$.

Example (n=2)

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix}$$

Tim Roughgarden

matrix 2

$$z_{ij} = \sum_{k=1}^{n} X_{ik} \cdot Y_{kj}$$

$\theta(n)$

Ter
vel

What is the asymptotic running time of the straightforward iterative algorithm for matrix multiplication?

- ○ $\theta(n \log n)$
- ○ $\theta(n^2)$
- ○ $\theta(n^3)$
- ○ $\theta(n^4)$

We assume that the time to access each entry is constant.

The whole idea of divide and conquer.

The Divide and Conquer Paradigm

1. DIVIDE into smaller subproblems
2. CONQUER subproblems recursively.
3. COMBINE solutions of subproblems into one for the original problem.

Tim Roughgarden

Applying Divide and Conquer to matrix multiplication.

Idea :

Write $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}_{n \times n}$ and $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}_{n \times n}$

[where A through H are all n/2 by n/2 matrices]

Then : (you check)

$$X \cdot Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

we express the original problem in terms of the multiplication $\frac{n}{2} \cdot \frac{n}{2}$ matrices

Tim Roughgarden

Recursive Algorithm #1

$AE + BG$ $AF + BH$
$CE + DG$ $CF + DH$

$AE, BG, AF, BH, CE, DG, CF, DH$

<u>Step 1</u> : recursively compute the 8 necessary products. of $\frac{n}{2} \cdot \frac{n}{2}$ matrices

<u>Step 2</u> : do the necessary additions $(\theta(n^2)\ time)$ ✓

<u>Fact</u> : runtime is $\theta(n^3)$ [follows from the master method]

Strassen's Algorithm (1969)

Step 1 : recursively compute only 7 (cleverly chosen) products

7 vs 8 seemingly not very significant, but since we do it over and over again the effects are fundam

Step 2 : do the necessary (clever) additions + subtractions (still $\theta(n^2)$ time)

Fact : better than cubic time!

[see Master Method lecture]

Tim Roughgarden

See the book

Is cost of summation and multiplication the same?

$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$

Is it really helping

The Details

The Seven Products : $P_1 = A(F-H)$, $P_2 = (A+B)H$, $P_3 = (C+D)E$, $P_4 = D(G-E)$, $P_5 = (A+D)(E+H)$, $P_6 = (B-D)(G+H)$, $P_7 = (A-C)(E+F)$

Claim : $X \cdot Y = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix} = \begin{pmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{pmatrix}$

Proof: $AE + \cancel{AH} + \cancel{DE} + \cancel{DH} + \cancel{DG} - \cancel{DE} - \cancel{AH} - \cancel{BH} + BG + \cancel{BH} - \cancel{DG} - \cancel{DH} = AE + BG$

Q.E.D

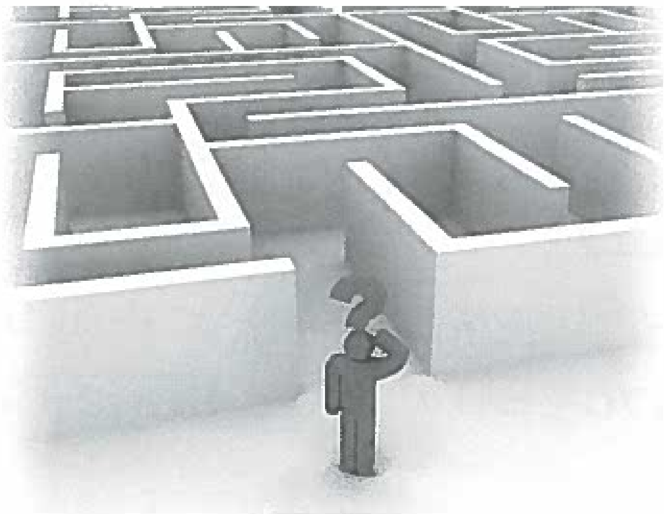
Question : where did this come from ? (remains open!)

Tim Roughgarden

matrix-5

cost of X and $\pm$

16



Design and Analysis
of Algorithms I

Divide and Conquer

Closest Pair I

. from computational geometry .
. advanced material .

pair - 1

The Closest Pair Problem

Input : a set $P = \{p_1, ..., p_n\}$ of n points in the plane R^2.

Notation : $d(p_i, p_j)$ = Euclidean distance

So if $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Output : a pair $p*, q* \in P$ of distinct points that minimize d(p,q) over p,q in the set P

Initial Observations

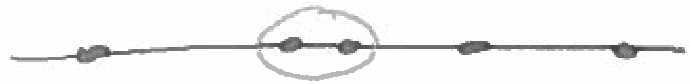
Assumption : (for convenience) all points have distinct x-coordinates, distinct y-coordinates. no ties.

two loops of size n (no. of points).

Brute-force search : takes $\theta(n^2)$ time.

For now, the easy problems

1-D Version of Closest Pair :



Steps

1. Sort points (O(nlog(n)) time)
2. Return closest pair of adjacent points (O(n) time)

$\Big\}$ O(n log n).

Can we get the same running time for the 2D version

Goal : O(nlog(n)) time algorithm for 2-D version.

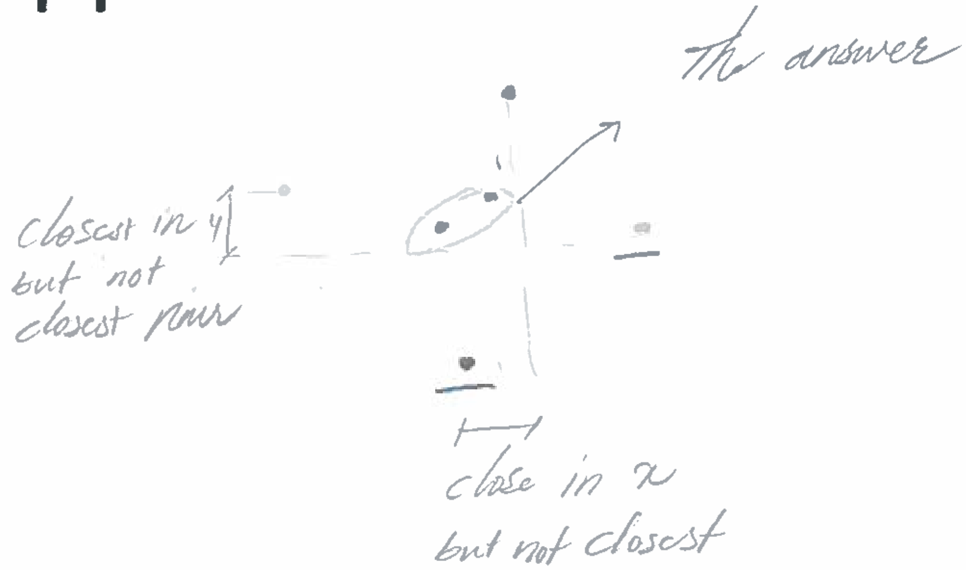
pair-2

High-Level Approach

1. Make copies of points sorted by x-coordinate (P_x) and by y-coordinate (P_y)
   [O(nlog(n)) time]

   (but this is not enough!)

2. Use Divide+Conquer



The Divide and Conquer Paradigm

1. DIVIDE into smaller subproblems.
2. CONQUER subproblems recursively.
3. COMBINE solutions of subproblems into one for the original problem. The genuity is here

Tim Roughgarden

Error in the slides and his teaching but the book is fine

pair - 3

we sort in the x and then cut in half.

ClosestPair(P_x, P_y)

$n \leq 3$ $O(1)$.

BASE CASE OMITTED

division step

1. Let Q = left half of P, R = right half of P. Form Q_x, Q_y, R_x, R_y [takes O(n) time] sorted by x coordinate and y coordinate

recursive calls

2. (p_1,q_1) = ClosestPair(Q_x,Q_y)
3. (p_2,q_2) = ClosestPair(R_x,R_y)

the lucky case is that the closest pair is either in Q or R

4. (p_3,q_3) = ClosestSplitPair(P_x,P_y) unlucky case
5. Return best of (p_1,q_1), (p_2,q_2), (p_3,q_3)

Tim Roughgarden

Key idea means that we dont need to find the closest split pair and then compare it with the closest of the left & right.

But, we go through it only if the closest split pair is closer than the other two cases.

Suppose we can correctly implement the ClosestSplitPair subroutine in O(n) time. What will be the overall running time of the Closest Pair algorithm ? (Choose the smallest upper bound that applies.)

- ○ $O(n)$
- ○ $O(n \log n)$
- ○ $O(n (\log n)^2)$
- ○ $O(n^2)$

we dont need full blown split pair

Key Idea : only need to bother computing the closest split pair in "unlucky case" where its distance is less than $d(p_1,q_1)$ and $d(p_2,q_2)$.

Result of 1st recursive call

Result of 2nd recursive call

only in the unlucky case we go through the split case.

δ says if we actually care about the split pair or not.

ClosestPair(P_x, P_y)

1. Let Q = left half of P, R = right half of P. Form Q_x, Q_y, R_x, R_y [takes O(n) time]
2. (p_1,q_1) = ClosestPair(Q_x,Q_y)
3. (p_2,q_2) = ClosestPair(R_x,R_y)
4. Let $\delta = min\{d(p_1,q_1), d(p_2,q_2)\}$
5. (p_3,q_3) = ClosestSplitPair(P_x,P_y,δ)
6. Return best of (p_1,q_1), (p_2,q_2), (p_3,q_3)

WILL DESCRIBE NEXT

BASE CASE OMITTED

The cost if we don't go through split.

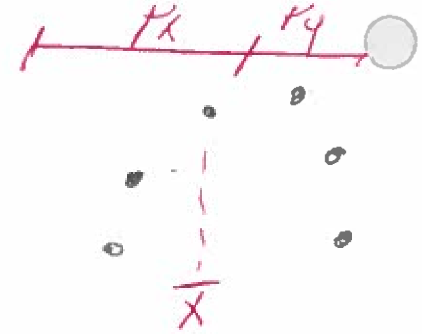
Requirements

1. O(n) time
2. Correct whenever closest pair of P is a split pair (we go through this only if we have the closest split pair)

The cost is highly non obvious.

Tim Roughgarden

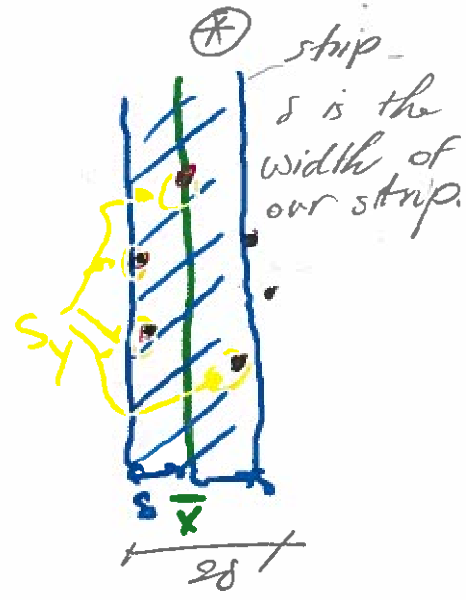
If a splitpair is the closest it should be in this strip.



ClosestSplitPair(P_x, P_y, δ)

Let $\bar{x}$ = biggest x-coordinate in left of P. (O(1) time)

Let S_y = points of P with x-coordinate ~~in~~ btw $\bar{x}-\delta$ and $\bar{x}+\delta$,
Sorted by y-coordinate (O(n) time)



Initialize best = δ, best pair = NULL

For i = 1 to $|S_y|$ - ~~7~~ 1

 For j = 1 to ~~7~~ min(7, ℓ-i)

  Let p,q = i^{th}, $(i+j)^{th}$ points of S_y

  If d(p,q) < best

   best pair = (p,q), best = d(p,q)

O(1) time

O(n) time

At end return best pair

Tim Roughgarden

Correctness Claim

$min\{d(p_1,q_1),d(p_2,q_2)\}$

<u>Claim</u> : Let $p \in Q, q \in R$ be a split pair with d(p,q) < δ

Then: (A) p and q are members of S_y
(B) p and q are at most 7 positions apart in S_y.

why

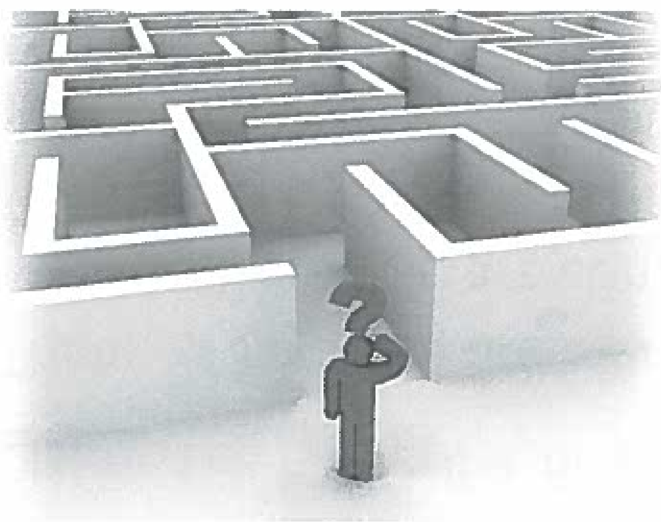
<u>Corollary1</u> : If the closest pair of P is a split pair, then the ClosestSplitPair finds it.

<u>Corollary2</u> ClosestPair is correct, and runs in O(nlog(n)) time.

Assuming claim is true!

Tim Roughgarden

July 20, 2018 (17)



Divide and Conquer

Closest Pair II

Design and Analysis of Algorithms I

pairII 1

CountSplitPair ≡ ClosestSplitPair

Correctness Claim

$min\{d(p_1,q_1),d(p_2,q_2)\}$

<u>Claim</u> : Let $p \in Q, q \in R$ be a split pair with $d(p,q) < \delta$

Then: (A) p and q are members of S_y

(B) p and q are at most 7 positions apart in S_y.

We want to prove this

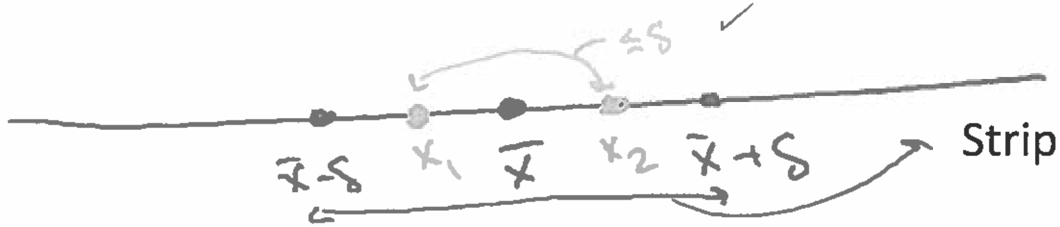
Proof of Correctness Claim (A)

from the left split — from the right side

Let $p = (x_1, y_1) \in Q,\ q = (x_2, y_2) \in R,\ d(p,q) \leq \delta$

<u>Note</u> : Since $d(p,q) \leq \delta,\ |x_1 - x_2| \leq \delta\ and\ |y_1 - y_2| \leq \delta$ correct

Proof of (A) [p and q are members of S_y i.e. $x_1, x_2 \in [\bar{x} - \delta, \bar{x} + \delta]$]



Strip that defines S_y

<u>Note</u> : $p \in Q \Rightarrow x_1 \leq \bar{x}\ and\ q \in R \Rightarrow x_2 \geq \bar{x}.$

=> $x_1, x_2 \in [\bar{x} - \delta, \bar{x} + \delta]$

pair II_2

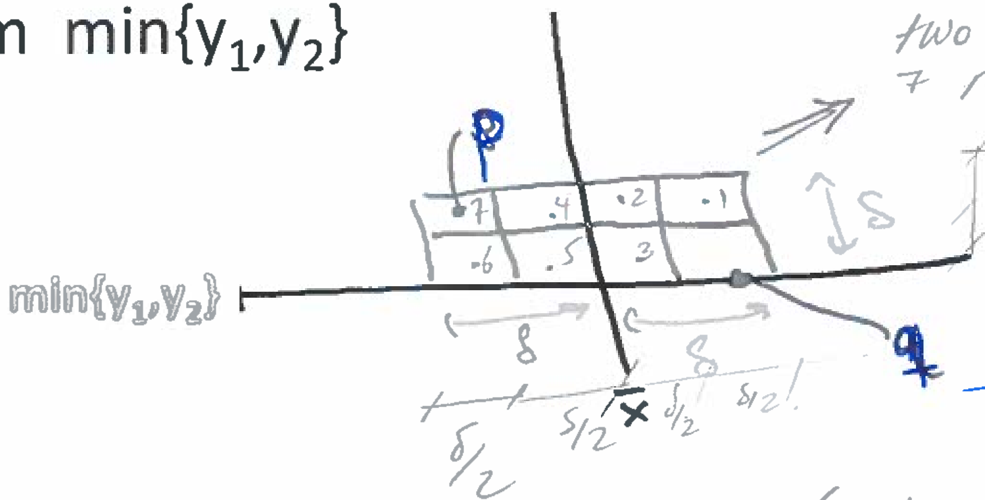
points where we assume $d(p,q) \leq \delta$

Proof of Correctness Claim (B)

(B) : p = (x_1,y_1) and q = (x_2,y_2) are at most 7 positions apart in S_y (indices in S_y differ by 7 positions.

Key Picture : draw $\delta/2 \times \delta/2$ boxes with center $\bar{x}$ and bottom $\min\{y_1,y_2\}$

two nodes are apart only 7 points (That's why we sort them in y-coordinate).



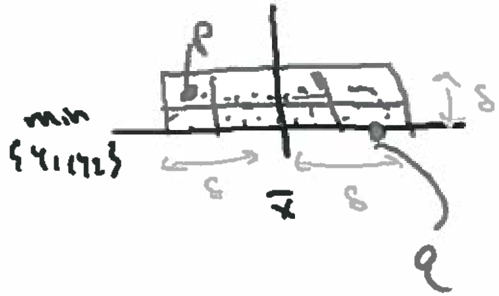
→ assume q has the lower y

Tim Roughgarden

see next page

$\boxed{\text{Assumption}}$ Sparse point. We assume each box has zero or one point why? bcs if we have more than one point in a box, it contradicts with the assumption that $\delta = \min\{d(l_1,l_2), d(r_1,r_2)\}$.

Proof of Correctness Claim (B)



Lemma 1 : all points of S_y with y-coordinate between those of p and q, inclusive, lie in one of these 8 boxes.

Proof : First, recall y-coordinates of p,q differ by $< \delta$

Second, by definition of S_y, all have x-coordinates between $\bar{x} - \delta$ *and* $\bar{x} + \delta$

Q.E.D

Tim Roughgarden

pair II–3

$d(a,b) = \sqrt{(\frac{\delta}{2})^2 + (\frac{\delta}{2})^2} = \delta\sqrt{\frac{1}{2}} = \frac{\delta}{\sqrt{2}} < \delta$

contradiction

δ supposed to be the smallest

Proof of Correctness Claim (B)

min $\{y_1, y_2\}$

Lemma 2 : At most one point of P in each box.

Proof : by contradiction

Suppose a,b lie in the same box. Then :

I. a,b are either both in Q or both in R ✓

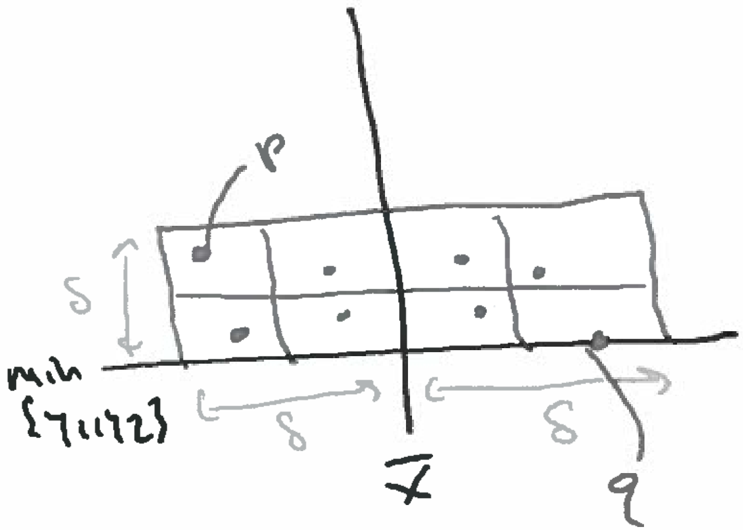
II. $d(a,b) \leq \frac{\delta}{2} \cdot \sqrt{2} \leq \delta$

But (i) and (ii) contradict the definition of δ (as smallest distance between pairs of points in Q or in R)

Q.E.D

Final Wrap-Up

Lemmas 1 and 2 => at most 8 points in this picture (including p and q)



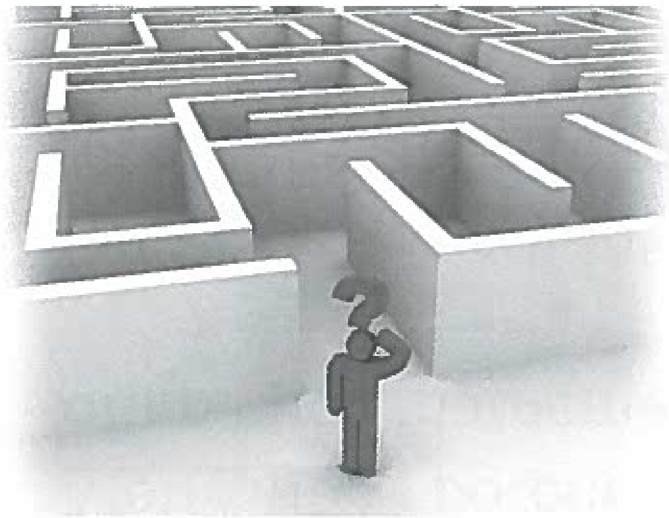
=> Positions of p,q in S_y differ by at most 7

Q.E.D

Tim Roughgarden

pair II 4

July 20, 2018 18



Design and Analysis
of Algorithms I

Master Method

Motivation

Integer Multiplication Revisited

Motivation : potentially useful algorithmic ideas often need mathematical analysis to evaluate

Recall : grade-school multiplication algorithm uses $\theta(n^2)$ operation to multiply two n-digit numbers ex $X = \underbrace{56}_{a}\underbrace{78}_{b} = \underbrace{56}_{a} \cdot 10^{\frac{4}{2}} + \underbrace{78}_{b}$ $*$ $y = 1234 = 12 * 10^{\frac{4}{2}} + 34$

A Recursive Algorithm

let's assume n is even.

Recursive approach

Write $x = 10^{n/2}a + b$ $y = 10^{n/2}c + d$

[where a,b,c,d are n/2 – digit numbers]

So :

$$x \cdot y = 10^n ac + 10^{n/2}(ad + bc) + bd \qquad (*)$$

Algorithm#1 : recursively compute ac,ad,bc,bd,
then compute (*) in the obvious way. and append zeros.

A Recursive Algorithm

T(n) = maximum number of operations this algorithm needs to multiply two n-digit numbers

Recurrence : express T(n) in terms of running time of recursive calls.

Every recurrence has two parts.

Base Case : T(1) <= a constant.

For all n > 1 :

$$T(n) \leq 4T(n/2) + O(n)$$

*1 Work done by recursive calls

*2 Work done here (outside of the recursive calls)

(*1) In case of integer multiplication there are 4 recursive calls.

(*2) means for examples padding the numbers and adding them up.

A Better Recursive Algorithm

Algorithm #2 (Gauss) : recursively compute ac[1], bd[2], (a+b)(c+d)[3] [recall ad+bc = (3) – (1) – (2)]

⟹ only 3 quantities for recursive calls.

New Recurrence :

Base Case : T(1) <= a constant

Ter
ver

Which recurrence best describes the running time of Gauss's algorithm for integer multiplication?

- ○ $T(n) \leq 2T(n/2) + O(n^2)$
- ○ $3T(n/2) + O(n)$
- ○ $4T(n/2) + O(n)$
- ○ $4T(n/2) + O(n^2)$

comparision with MergeSort: we only have 2 recursive calls. but outside of the recursive calls, the merge part, has linear own cost.

A Better Recursive Algorithm

Algorithm #2 (Gauss) : recursively compute ac[(1)], bd[(2)], (a+b)(c+d)[(3)] [recall ad+bc = (3) – (1) – (2)]

New Recurrence :

Base Case : T(1) <= a constant

For all n>1 : $T(n) \leq 3T(n/2) + O(n)$

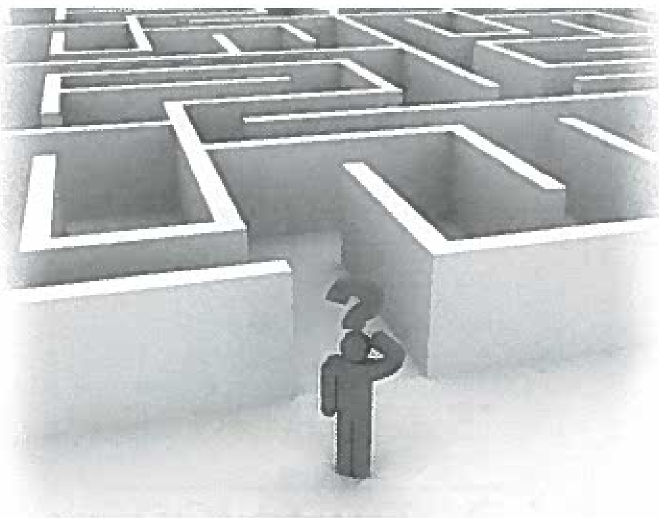
Work done here (linear work).

Work done by recursive calls

⟹ this would be better than $4T(\frac{n}{2})$.

only 3 recursive calls

July 29, 18 19



Design and Analysis of Algorithms I

Master Method

The Precise Statement

Used to analyse recursive algo.

= upper bound on the recursive algo.

= running time.

The Master Method

Cool Feature : a "black box" for solving recurrences.

Assumption : all subproblems have equal size. $(\frac{n}{2})$

If the sizes are different, you cannot ~~use~~ the master method.

Tim Roughgarden

Two ingredients of each master algo.

Recurrence Format

1. Base Case : T(n) <= a constant for all sufficiently small n
2. For all larger n :

We ignore constants in big oh notation but not

$$T(n) \le aT(n/b) + O(n^d)$$

where

a = number of recursive calls (>= 1)

b = input size shrinkage factor (> 1) better to be greater than one

d = exponent in running time of "combine step" (>=0)

[a,b,d independent of n]

should be

The Master Method

(Or master theorem)

gives only bigger bounds (worse case).

Base doesn't matter (only changes leading constants)

The running time is bounded by three things

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

dominated but outside recursion

Base matters (because that's in the exponent.) matters a lot.

different by a constant.

$\lg_2 n = C \lg_e n$

WIKIPEDIA

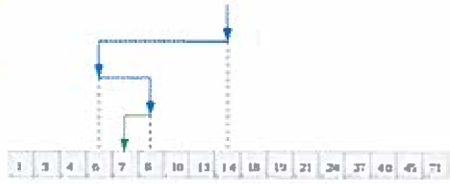
Binary search algorithm

In computer science, **binary search**, also known as **half-interval search**,[1] **logarithmic search**,[2] or **binary chop**,[3] is a search algorithm that finds the position of a target value within a sorted array.[4][5] Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array. Even though the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

Binary search runs in at worst logarithmic time, making $O(\log n)$ comparisons, where n is the number of elements in the array, the O is Big O notation, and $\log$ is the logarithm. Binary search takes constant ($O(1)$) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.[6] Binary search is faster than linear search except for small arrays, but the array must be sorted first. Although specialized data structures designed for fast searching—such as hash tables—can be searched more efficiently, binary search applies to a wider range of problems.

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays, efficiently solving a series of search problems in computational geometry and numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data structures are based on binary search.

Binary search algorithm



Visualization of the binary search algorithm where 7 is the target value.

Class	Search algorithm
Data structure	Array
Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$

$n > \lg n$

Contents

Algorithm

Binary search works on sorted arrays. Binary search begins by comparing the middle element of the array with the target value. If the target value matches the middle element, its position in the array is returned. If the target value is less than or greater than the middle element, the search continues in the lower or upper half of the array, respectively, eliminating the other half from consideration.[7]

Procedure

Given an array A of n elements with values or records $A_0, A_1, \ldots, A_{n-1}$, sorted such that $A_0 \leq A_1 \leq \ldots \leq A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of T in A.[7]

1. Set L to 0 and R to $n - 1$.
2. If $L > R$, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor, or the greatest integer less than $(L + R)/2$.
4. If $A_m < T$, set L to $m + 1$ and go to step 2.
5. If $A_m > T$, set R to $m - 1$ and go to step 2.
6. Now $A_m = T$, the search is done; return m.

This iterative procedure keeps track of the search boundaries with the two variables. The procedure may be expressed in pseudocode as follows, where the variable names and types remain the same as above, `floor` is the floor function, and `unsuccessful` refers to a specific variable that conveys the failure of the search.[7]

```
function binary_search(A, n, T):
    L := 0
    R := n - 1
    while L <= R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else if A[m] > T:
            R := m - 1
        else:
            return m
    return unsuccessful
```

Alternative procedure

In the above procedure, the algorithm checks whether the middle element (m) is equal to the target (T) in every iteration. Some implementations leave out this check during each iteration. The algorithm would perform this check only when one element is left (when $L = R$). This results in a faster comparison loop, as one comparison is eliminated per iteration. However, it requires one more iteration on average.[8]

The first implementation to leave out this equality check was published by Hermann Bottenbruch in 1962. However, Bottenbruch's version assumes that there is more than one element in the array. The subroutine for Bottenbruch's implementation is as follows, assuming that $n > 1$:[9][8]

1. Set L to 0 and R to $n - 1$.
2. If $L \geq R$, go to step 6.
3. Set m (the position of the middle element) to the ceiling, or the least integer greater than $(L + R)/2$.
4. If $A_m > T$, set R to $m - 1$ and go to step 2.
5. Otherwise, if $A_m \leq T$, set L to m and go to step 2.
6. Now $L = R$, the search is done. If $A_L = T$, return L. Otherwise, the search terminates as unsuccessful.

Duplicate elements

The procedure may return any index whose element is equal to the target value, even if there are duplicate elements in the array. For example, if the array to be searched was $[1, 2, 3, 4, 4, 5, 6, 7]$ and the target was 4, then it would be correct for the algorithm to either return the 4th (index 3) or 5th (index 4) element. The regular procedure would return the 4th element (index 3). However, it is sometimes necessary to find the leftmost element or the rightmost element if the target value is duplicated in the array. In the above example, the 4th element is the leftmost element of the value 4, and the 5th element is the rightmost element. The alternative procedure above will always return the index of the rightmost element if an element is duplicated in the array.[9]

Procedure for finding the leftmost element

To find the leftmost element, the following procedure can be used:[10]

1. Set L to 0 and R to n.
2. If $L \geq R$, go to step 6.
3. Set m (the position of the middle element) to the floor, or the greatest integer less than $(L + R)/2$.
4. If $A_m < T$, set L to $m + 1$ and go to step 2.
5. Otherwise, if $A_m \geq T$, set R to m and go to step 2.
6. Now $L = R$, the search is done, return L.

If $L < n$ and $A_L = T$, then A_L is the leftmost element that equals T. Even if T is not in the array, L is the rank of T in the array, or the number of elements in the array that are less than T.

Where `floor` is the floor function and == checks for equality, the pseudocode for this version is:

```
function binary_search_leftmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else:
            R := m
    return L
```

Procedure for finding the rightmost element

To find the rightmost element, the following procedure can be used:[10]

1. Set L to 0 and R to n.
2. If $L \geq R$, go to step 6.
3. Set m (the position of the middle element) to the floor, or the greatest integer less than $(L + R)/2$.
4. If $A_m > T$, set R to m and go to step 2.
5. Otherwise, if $A_m \leq T$, set L to $m + 1$ and go to step 2.
6. Now $L = R$, the search is done, return L.

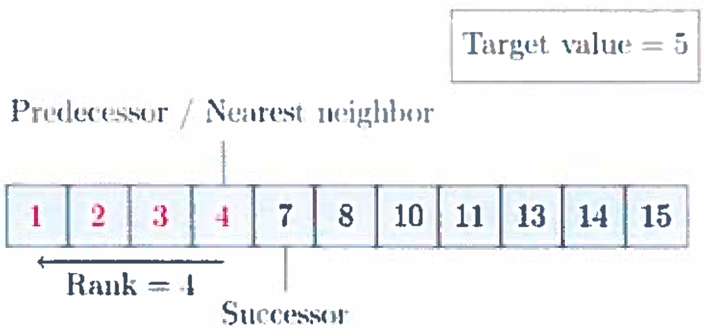
If $L < n$ and $A_L = T$, then A_L is the rightmost element that equals T. Even if T is not in the array, L is the number of elements in the array that are greater than T.

Where `floor` is the floor function and == checks for equality, the pseudocode for this version is:

```
function binary_search_rightmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] > T:
            R := m
        else:
            L := m + 1
    return L
```

Approximate matches

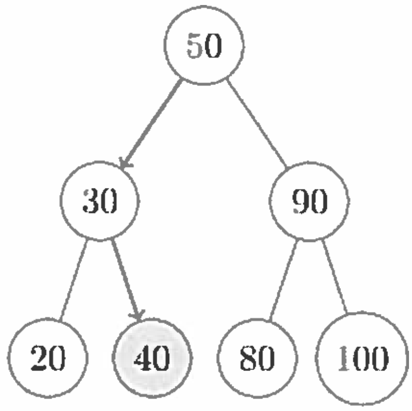
The above procedure only performs *exact* matches, finding the position of a target value. However, due to the ordered nature of sorted arrays, it is trivial to extend binary search to perform approximate matches. For example, binary search can be used to compute, for a given value, its rank (the number of smaller elements), predecessor (next-smallest element), successor (next-largest element), and nearest neighbor. Range queries seeking the number of elements between two values can be performed with two rank queries.[11]



Binary search can be adapted to compute approximate matches. In the example above, the rank, predecessor, successor, and nearest neighbor are shown for the target value 5, which is not in the array.

- Rank queries can be performed using a modified version of binary search. By returning m on a successful search, and L on an unsuccessful search, the number of elements *less than* the target value is returned instead.[11] If there are duplicate elements in the array, then the procedure for finding the leftmost element must be used.
- Predecessor and successor queries can be performed with rank queries. Once the rank of the target value is known, its predecessor is the element at the position given by its rank (as it is the largest element that is smaller than the target value). Its successor is the element after it (if it is present in the array) or at the next position after the predecessor (otherwise).[12] The procedure for finding the rightmost element must be used for computing the successor if there are duplicate elements in the array. The nearest neighbor of the target value is either its predecessor or successor, whichever is closer.
- Range queries are also straightforward. Once the ranks of the two values are known, the number of elements greater than or equal to the first value and less than the second is the difference of the two ranks. This count can be adjusted up or down by one according to whether the endpoints of the range should be considered to be part of the range and whether the array contains keys matching those endpoints.[13]

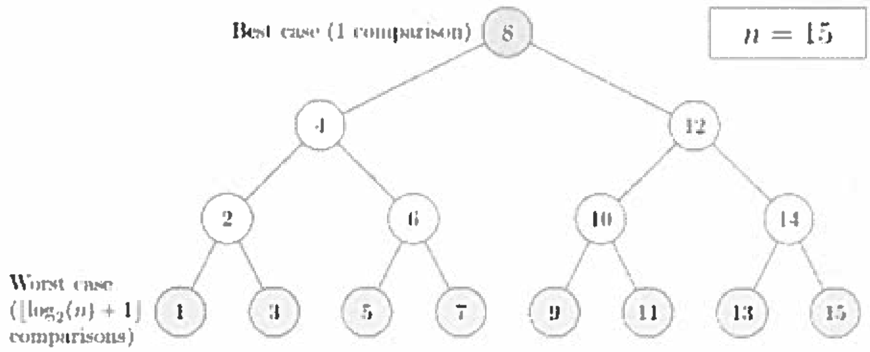
Performance



A tree representing binary search. The array being searched here is [20, 30, 40, 50, 80, 90, 100], and the target value is 40.

The performance of binary search can be analyzed by reducing the procedure to a binary comparison tree, where the root node is the middle element of the array. The middle element of the lower half is the left child node of the root and the middle element of the upper half is the right child node of the root. The rest of the tree is built in a similar fashion. This model represents binary search; starting from the root node, the left or right subtrees are traversed depending on whether the target value is less or more than the node under consideration, representing the successive elimination of elements.[6][14]

The worst case is iterations of the comparison loop, where the notation denotes the floor function that rounds its argument to the next-smallest integer and is the binary logarithm. The worst case is reached when the search reaches the deepest level of the tree, equivalent to a binary search that has reduced to one element and, in each iteration, always eliminates the smaller subarray out of the two if they are not of equal size.[a][14]



The worst case is reached when the search reaches the deepest level of the tree, while the best case is reached when the target value is the middle element.

The worst case may also be reached when the target element is not in the array. If is one less than a power of two, then this is always the case. Otherwise, the search may perform , the worst case, or iterations, one less than the worst case, depending on whether the search reaches the deepest or second-deepest level of the tree.[15]

On average, assuming that each element is equally likely to be searched, binary search makes iterations when the target element is in the array. This is approximately equal to iterations. When the target element is not in the array, the average case is iterations, assuming that the range between and outside elements is equally likely to be searched.[14]

In the best case, where the target value is the middle element of the array, its position is returned after one iteration.[16]

In terms of iterations, no search algorithm that works only by comparing elements can exhibit better average and worst-case performance than binary search. This is because the comparison tree representing binary search has the fewest levels possible as each level is filled completely with nodes if there are enough nodes.[b] Otherwise, the search algorithm can eliminate few elements in an iteration, increasing the number of iterations required in the average and worst case. This is the case for other search algorithms based on comparisons, as while they may work faster on some target values, the average performance over *all* elements is affected. By dividing the array in half, binary search ensures that the size of both subarrays are as similar as possible.[14]

Performance of alternative procedure

Each iteration of the binary search procedure defined above makes one or two comparisons, checking if the middle element is equal to the target in each iteration. Assuming that each element is equally likely to be searched, each iteration makes 1.5 comparisons on average. A variation of the algorithm checks whether the middle element is equal to the target at the end of the search, eliminating on average half a comparison from each iteration. This slightly cuts the time taken per iteration on most computers, while guaranteeing that the search takes the maximum number of iterations, on average adding one iteration to the search. Because the comparison loop is performed only times in the worst case, the slight increase in comparison loop efficiency does not compensate for the extra iteration for all but enormous .[c][17][18]

Binary search versus other schemes

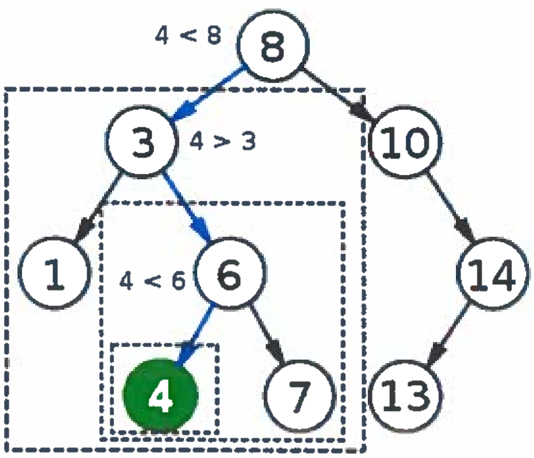
Sorted arrays with binary search are a very inefficient solution when insertion and deletion operations are interleaved with retrieval, taking time for each such operation, and complicating memory use.[19] There are other data structures that support much more efficient insertion and deletion. Binary search can be used to perform exact matching and set membership (determining whether a target value is in a collection of values), but there are data structures that support faster exact matching and set membership. However, unlike many other searching schemes, binary search can be used for efficient approximate matching, usually performing such matches in time regardless of the type or structure of the values themselves.[20] In addition, there are some operations, like finding the smallest and largest element, that can be performed efficiently on a sorted array.[11]

Hashing

For implementing associative arrays, hash tables, a data structure that maps keys to records using a hash function, are generally faster than binary search on a sorted array of records;[21] most implementations require only amortized constant time on average.[d][23] However, hashing is not useful for approximate matches, such as computing the next-smallest, next-largest, and nearest key, as the only information given on a failed search is that the target is not present in any record.[24] Binary search is ideal for such matches, performing them in logarithmic time. Binary search also supports approximate matches. Some operations, like finding the smallest and largest element, can be done efficiently on sorted arrays but not on hash tables.[20]

Trees

A binary search tree is a binary tree data structure that works based on the principle of binary search. The records of the tree are arranged in sorted order, and each record in the tree can be searched using an algorithm similar to binary search, taking on average logarithmic time. Insertion and deletion also require on average logarithmic time in binary search trees. This can be faster than the linear time insertion and deletion of sorted arrays, and binary trees retain the ability to perform all the operations possible on a sorted array, including range and approximate queries.[20][25]



Binary search trees are searched using an algorithm similar to binary search.

However, binary search is usually more efficient for searching as binary search trees will most likely be imperfectly balanced, resulting in slightly worse performance than binary search. This even applies to balanced binary search trees, binary search trees that balance their own nodes, because they rarely produce *optimally*-balanced trees. Although unlikely, the tree may be severely imbalanced with few internal nodes with two children, resulting in the average and worst-case search time approaching comparisons.[e] Binary search trees take more space than sorted arrays.[27]

Binary search trees lend themselves to fast searching in external memory stored in hard disks, as binary search trees can efficiently be structured in filesystems. The B-tree generalizes this method of tree organization; B-trees are frequently used to organize long-term storage such as databases and filesystems.[28][29]

Linear search

Linear search is a simple search algorithm that checks every record until it finds the target value. Linear search can be done on a linked list, which allows for faster insertion and deletion than an array. Binary search is faster than linear search for sorted arrays except if the array is short, although the array needs to be sorted beforehand.[f][31] All sorting algorithms based on comparing elements, such as quicksort and merge sort, require at least comparisons in the worst case.[32] Unlike linear search, binary search can be used for efficient approximate matching. There are operations such as finding the smallest and largest element that can be done efficiently on a sorted array but not on an unsorted array.[33]

Set membership algorithms

A related problem to search is set membership. Any algorithm that does lookup, like binary search, can also be used for set membership. There are other algorithms that are more specifically suited for set membership. A bit array is the simplest, useful when the range of keys is limited. It compactly stores a collection of bits, with each bit representing a single key within the range of keys. Bit arrays are very fast, requiring only time.[34] The Judy1 type of Judy array handles 64-bit keys efficiently.[35]

For approximate results, Bloom filters, another probabilistic data structure based on hashing, store a set of keys by encoding the keys using a bit array and multiple hash functions. Bloom filters are much more space-efficient than bit arrays in most cases and not much slower: with hash functions, membership queries require only time. However, Bloom filters suffer from false positives.[g][h][37]

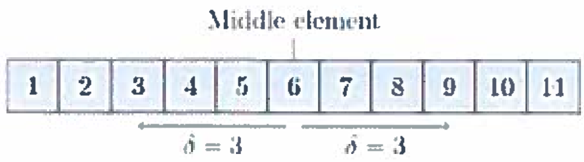
Other data structures

There exist data structures that may improve on binary search in some cases for both searching and other operations available for sorted arrays. For example, searches, approximate matches, and the operations available to sorted arrays can be performed more efficiently than binary search on specialized data structures such as van Emde Boas trees, fusion trees, tries, and bit arrays. However, while these operations can always be done at least efficiently on a sorted array regardless of the keys, such data structures are usually only faster because they exploit the properties of keys with a certain attribute (usually keys that are small integers), and thus will be time or space consuming for keys that lack that attribute.[20] Some structures, such as Judy arrays, use a combination of approaches to mitigate this while retaining efficiency and the ability to perform approximate matching.[35]

Variations

Uniform binary search

Uniform binary search stores, instead of the lower and upper bounds, the index of the middle element and the change in the middle element from the current iteration to the next iteration. Each step reduces the change by about half. For example, if the array to be searched was [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], the middle element would be 6. Uniform binary search works on the basis that the difference between the index of middle element of the array and the left and right subarrays is the same. In this case, the middle element of the left subarray ([1, 2, 3, 4, 5]) is 3 and the middle element of the right subarray ([7, 8, 9, 10, 11]) is 9. Uniform binary search would store the value of 3 as both indices differ from 6 by this same amount.[38] To reduce the search space, the algorithm either adds or subtracts this change from the middle element. The main advantage of uniform binary search is that the procedure can store a table of the differences between indices for each iteration of the procedure. Uniform binary search may be faster on systems where it is inefficient to calculate the midpoint, such as on decimal computers.[39]



Uniform binary search stores the difference between the current and the two next possible middle elements instead of specific bounds.

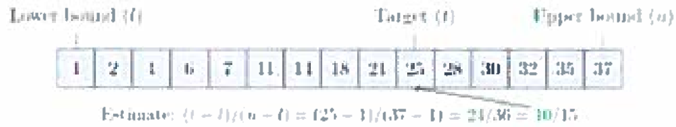
Exponential search

Exponential search extends binary search to unbounded lists. It starts by finding the first element with an index that is both a power of two and greater than the target value. Afterwards, it sets that index as the upper bound, and switches to binary search. A search takes iterations of the exponential search and at most iterations of the binary search, where x is the position of the target value. Exponential search works on bounded lists, but becomes an improvement over binary search only if the target value lies near the beginning of the array.[40]

Visualization of exponential searching finding the upper bound for the subsequent binary search.

Interpolation search

Instead of calculating the midpoint, interpolation search estimates the position of the target value, taking into account the lowest and highest elements in the array as well as length of the array. This is only possible if the array elements are numbers. It works on the basis that the midpoint is not the best guess in many cases. For example, if the target value is close to the highest element in the array, it is likely to be located near the end of the array.[41] When the distribution of the array elements is uniform or near uniform, it makes comparisons.[41][42][43]



Visualization of interpolation search. In this case, no searching is needed because the estimate of the target's location within the array is correct. Other implementations may specify another function for estimating the target's location.

In practice, interpolation search is slower than binary search for small arrays, as interpolation search requires extra computation. Although its time complexity grows more slowly than binary search, this only compensates for the extra computation for large arrays.[41]

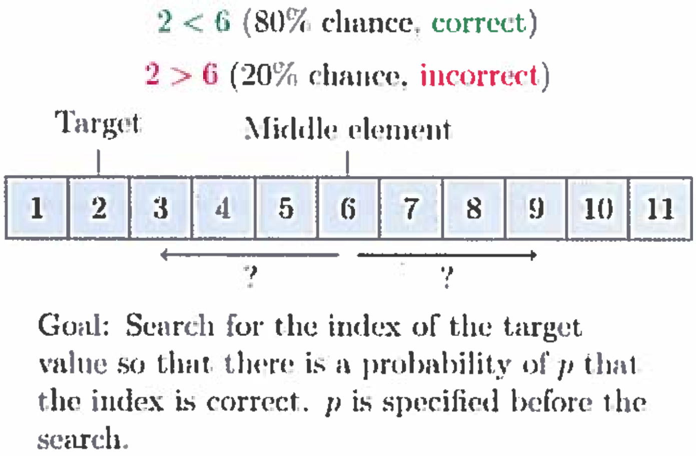
Fractional cascading

Fractional cascading is a technique that speeds up binary searches for the same element in multiple sorted arrays. Searching each array separately requires time, where is the number of arrays. Fractional cascading reduces this to by storing specific information in each array about each element and its position in the other arrays.[44][45]

Fractional cascading was originally developed to efficiently solve various computational geometry problems, but it also has been applied elsewhere, in domains such as data mining and Internet Protocol routing.[44]

Noisy binary search

Noisy binary search algorithms solve the case where the algorithm cannot reliably compare elements of the array. For each pair of elements, there is a certain probability that the algorithm makes the wrong comparison. Noisy binary search can find the correct position of the target with a given probability that controls the reliability of the yielded position.[l][j][49][50]



In noisy binary search, there is a certain probability that a comparison is incorrect.

Quantum binary search

Classical computers are bounded to the worst case of exactly iterations when performing binary search. Quantum algorithms for binary search are still bounded to a proportion of queries (representing iterations of the classical procedure), but the constant factor is less than one, providing for faster performance on quantum computers. Any *exact* quantum binary search procedure—that is, a procedure that always yields the correct result—requires at least – queries in the worst case, where is the natural logarithm.[51] There is an exact quantum binary search procedure that runs in queries in the worst case.[52] In comparison, Grover's algorithm is the optimal quantum algorithm for searching an unordered list of elements, and it requires – queries.[53]

History

In 1946, John Mauchly made the first mention of binary search as part of the Moore School Lectures, a seminal and foundational college course in computing.[9] In 1957, William Wesley Peterson published the first method for interpolation search.[9][54] Every published binary search algorithm worked only for arrays whose length is one less than a power of two[k] until 1960, when Derrick Henry Lehmer published a binary search algorithm that worked on all arrays.[56] In 1962, Hermann Bottenbruch presented an ALGOL 60 implementation of binary search that placed the comparison for equality at the end, increasing the average number of iterations by one, but reducing to one the number of comparisons per iteration.[8] The uniform binary search was developed by A. K. Chandra of Stanford University in 1971.[9] In 1986, Bernard Chazelle and Leonidas J. Guibas introduced fractional cascading as a method to solve numerous search problems in computational geometry.[44][57][58]

Implementation issues

> Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky ... — Donald Knuth[2]

When Jon Bentley assigned binary search as a problem in a course for professional programmers, he found that ninety percent failed to provide a correct solution after several hours of working on it, mainly because the incorrect implementations failed to run or returned a wrong answer in rare edge cases.[59] A study published in 1988 shows that accurate code for it is only found in five out of twenty textbooks.[60] Furthermore, Bentley's own implementation of binary search, published in his 1986 book *Programming Pearls*, contained an overflow error that remained undetected for over twenty years. The Java programming language library implementation of binary search had the same overflow bug for more than nine years.[61]

In a practical implementation, the variables used to represent the indices will often be of fixed size, and this can result in an arithmetic overflow for very large arrays. If the midpoint of the span is calculated as $(L + R)/2$, then the value of $L + R$ may exceed the range of integers of the data type used to store the midpoint, even if L and R are within the range. If L and R are nonnegative, this can be avoided by calculating the midpoint as $L + (R - L)/2$.[62]

If the target value is greater than the greatest value in the array, and the last index of the array is the maximum representable value of L, the value of L will eventually become too large and overflow. A similar problem will occur if the target value is smaller than the least value in the array and the first index of the array is the smallest representable value of R. In particular, this means that R must not be an unsigned type if the array starts with index 0.[60][62]

An infinite loop may occur if the exit conditions for the loop are not defined correctly. Once L exceeds R, the search has failed and must convey the failure of the search. In addition, the loop must be exited when the target element is found, or in the case of an implementation where this check is moved to the end, checks for whether the search was successful or failed at the end must be in place. Bentley found that most of the programmers who incorrectly implemented binary search made an error in defining the exit conditions.[8][63]

Library support

Many languages' standard libraries include binary search routines:

- C provides the function `bsearch()` in its standard library, which is typically implemented via binary search, although the official standard does not require it so.[64]
- C++'s Standard Template Library provides the functions `binary_search()`, `lower_bound()`, `upper_bound()` and `equal_range()`.[65]
- COBOL provides the `SEARCH ALL` verb for performing binary searches on COBOL ordered tables.[66]
- Go's `sort` standard library package contains the functions `Search`, `SearchInts`, `SearchFloat64s`, and `SearchStrings`, which implement general binary search, as well as specific implementations for searching slices of integers, floating-point numbers, and strings, respectively.[67]
- Java offers a set of overloaded `binarySearch()` static methods in the classes `Arrays` (https://docs.oracle.com/javase/10/docs/api/java/util/Arrays.html) and `Collections` (https://docs.oracle.com/javase/10/docs/api/java/util/Collections.html) in the standard `java.util` package for performing binary searches on Java arrays and on `List`s, respectively.[68][69]
- Microsoft's .NET Framework 2.0 offers static generic versions of the binary search algorithm in its collection base classes. An example would be `System.Array`'s method `BinarySearch<T>(T[] array, T value)`.[70]
- For Objective-C, the Cocoa framework provides the `NSArray -indexOfObject:inSortedRange:options:usingComparator:` (https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/NSArray.html#//apple_ref/occ/instm/NSArray/indexOfObject:inSortedRange:options:usingComparator:) method in Mac OS X 10.6+.[71] Apple's Core Foundation C framework also contains a `CFArrayBSearchValues()` (https://developer.apple.com/library/mac/documentation/CoreFoundation/Reference/CFArrayRef/Reference/reference.html#//apple_ref/c/func/CFArrayBSearchValues) function.[72]
- Python provides the `bisect` module.[73]
- Ruby's Array class includes a `bsearch` method with built-in approximate matching.[74]

See also

- Bisection method – the same idea used to solve equations in the real numbers
- Multiplicative binary search - binary search variation with simplified midpoint calculation

Notes and references

Notes

a. This happens as binary search will not always divide the array perfectly. Take for example the array $[1, 2, ..., 16]$. The first iteration will select the midpoint of 8. On the left subarray are eight elements, but on the right are nine. If the search takes the right path, there is a higher chance that the search will make the maximum number of comparisons.[14]

b. Any search algorithm based solely on comparisons can be represented using a binary comparison tree. An *internal path* is any path from the root to an existing node. Let I be the *internal path length*, the sum of the lengths of all internal paths. If each element is equally likely to be searched, the average case is — or simply one plus the average of all the internal path lengths of the tree. This is because internal paths represent the elements that the search algorithm compares to the target. The lengths of these internal paths represent the number of iterations *after* the root node. Adding the average of these lengths to the one iteration at the root yields the average case. Therefore, to minimize the average number of comparisons, the internal path length must be minimized. It turns out that the tree for binary search minimizes the internal path length. Knuth 1998 proved that the *external path* length (the path length over all nodes where both children are present for each already-existing node) is minimized when the external nodes (the nodes with no children) lie within two consecutive levels of the tree. This also applies to internal paths as internal path length I is linearly related to external path length . For any tree of nodes, . When each subtree has a similar number of nodes, or equivalently the array is divided into halves in each iteration, the external nodes as well as their interior parent nodes lie within two levels. It follows that binary search minimizes the number of average comparisons as its comparison tree has the lowest possible internal path length.[14]

c. Knuth 1998 showed on his MIX computer model, intended to represent an ordinary computer, that the average running time of this variation for a successful search is units of time compared to units for regular binary search. The time complexity for this variation grows slightly more slowly, but at the cost of higher initial complexity. [17]

d. It is possible to perform hashing in guaranteed constant time.[22]

e. The *worst* binary search tree for searching can be produced by inserting the values in sorted order or in an alternating lowest-highest key pattern.[26]

f. Knuth 1998 performed a formal time performance analysis of both of these search algorithms. On Knuth's MIX computer, which Knuth designed as a representation of an ordinary computer, binary search takes on average units of time for a successful search, while linear search with a sentinel node at the end of the list takes — units. Linear search has lower initial complexity because it requires minimal computation, but it quickly outgrows binary search in complexity. On the MIX computer, binary search only outperforms linear search with a sentinel if .[14][30]

g. This is because simply setting all of the bits which the hash functions point to for a specific key can affect queries for other keys which have a common hash location for one or more of the functions.[36]

h. There exist improvements of the Bloom filter which improve on its complexity or support deletion; for example, the cuckoo filter exploits cuckoo hashing to gain these advantages.[36]

i. Using noisy comparisons, Ben-Or & Hassidim 2008 established that any noisy binary search procedure must make at least — — comparisons on average, where is the binary entropy function and is the probability that the procedure yields the wrong position.[46]

j. The noisy binary search problem can be considered as a case of the Rényi-Ulam game,[47] a variant of Twenty Questions where the answers may be wrong.[48]

k. That is, arrays of length 1, 3, 7, 15, 31 ...[55]

Citations

1. Willams, Jr., Louis F. (22 April 1976). *A modification to the half-interval search (binary search) method* (https://dl.acm.org/citation.cfm?doid=503561.503582). Proceedings of the 14th ACM Southeast Conference. ACM. pp. 95–101. doi:10.1145/503561.503582 (https://doi.org/10.1145/503561.503582). Retrieved 29 June 2018.

2. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Binary search".
3. Butterfield & Ngondi 2016, p. 46.
4. Cormen et al. 2009, p. 39.
5. Weisstein, Eric W. "Binary search" (http://mathworld.wolfram.com/BinarySearch.html). *MathWorld*.
6. Flores, Ivan; Madpis, George (1 September 1971). "Average binary search length for dense ordered lists" (https://dl.acm.org/citation.cfm?doid=362663.362752). *Communications of the ACM*. **14** (9): 602–603. doi:10.1145/362663.362752 (https://doi.org/10.1145/362663.362752). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782). Retrieved 29 June 2018.
7. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Algorithm B".
8. Bottenbruch, Hermann (1 April 1962). "Structure and use of ALGOL 60" (https://dl.acm.org/citation.cfm?doid=321119.321120). *Journal of the ACM (JACM)*. **9** (2): 161–221. doi:10.1145/321119.321120 (https://doi.org/10.1145/321119.321120). ISSN 0004-5411 (https://www.worldcat.org/issn/0004-5411). Retrieved 30 June 2018. Procedure is described at p. 214 (§43), titled "Program for Binary Search".
9. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "History and bibliography".
10. Kasahara & Morishita 2006, pp. 8–9.
11. Sedgewick & Wayne 2011, §3.1, subsection "Rank and selection".
12. Goldman & Goldman 2008, pp. 461–463.
13. Sedgewick & Wayne 2011, §3.1, subsection "Range queries".
14. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Further analysis of binary search".
15. Knuth 1998, §6.2.1 ("Searching an ordered table"), "Theorem B".
16. Chang 2003, p. 169.
17. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Exercise 23".
18. Rolfe, Timothy J. (1997). "Analytic derivation of comparisons in binary search". *ACM SIGNUM Newsletter*. **32** (4): 15–19. doi:10.1145/289251.289255 (https://doi.org/10.1145/289251.289255).
19. Knuth 1997, §2.2.2 ("Sequential Allocation").
20. Beame, Paul; Fich, Faith E. (2001). "Optimal bounds for the predecessor problem and related problems" (http://www.sciencedirect.com/science/article/pii/S0022000002918222). *Journal of Computer and System Sciences*. **65** (1): 38–72. doi:10.1006/jcss.2002.1822 (https://doi.org/10.1006/jcss.2002.1822).
21. Knuth 1998, §6.4 ("Hashing").
22. Knuth 1998, §6.4 ("Hashing"), subsection "History".
23. Dietzfelbinger, Martin; Karlin, Anna; Mehlhorn, Kurt; Meyer auf der Heide, Friedhelm; Rohnert, Hans; Tarjan, Robert E. (August 1994). "Dynamic perfect hashing: upper and lower bounds". *SIAM Journal on Computing*. **23** (4): 738–761. doi:10.1137/S0097539791194094 (https://doi.org/10.1137/S0097539791194094).
24. Morin, Pat. "Hash tables" (http://cglab.ca/~morin/teaching/5408/notes/hashing.pdf) (PDF). p. 1. Retrieved 28 March 2016.
25. Sedgewick & Wayne 2011, §3.2 ("Binary Search Trees"), subsection "Order-based methods and deletion".
26. Knuth 1998, §6.2.2 ("Binary tree searching"), subsection "But what about the worst case?".
27. Sedgewick & Wayne 2011, §3.5 ("Applications"), "Which symbol-table implementation should I use?".
28. Knuth 1998, §5.4.9 ("Disks and Drums").
29. Knuth 1998, §6.2.4 ("Multiway trees").
30. Knuth 1998, Answers to Exercises (§6.2.1) for "Exercise 5".
31. Knuth 1998, §6.2.1 ("Searching an ordered table").
32. Knuth 1998, §5.3.1 ("Minimum-Comparison sorting").
33. Sedgewick & Wayne 2011, §3.2 ("Ordered symbol tables").
34. Knuth 2011, §7.1.3 ("Bitwise Tricks and Techniques").

35. Silverstein, Alan, *Judy IV shop manual* (http://judy.sourceforge.net/doc/shop_interm.pdf) (PDF), Hewlett-Packard, pp. 80–81
36. Fan, Bin; Andersen, Dave G.; Kaminsky, Michael; Mitzenmacher, Michael D. (2014). *Cuckoo filter: practically better than Bloom*. Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. pp. 75–88. doi:10.1145/2674005.2674994 (https://doi.org/10.1145/2674005.2674994).
37. Bloom, Burton H. (1970). "Space/time trade-offs in hash coding with allowable errors" (https://web.archive.org/web/20041104063826/http://ovmj.org:80/gnunet/papers/p422-bloom.pdf) (PDF). *Communications of the ACM*. **13** (7): 422–426. doi:10.1145/362686.362692 (https://doi.org/10.1145/362686.362692). Archived from the original (http://www.ovmj.org/GNUnet/papers/p422-bloom.pdf) (PDF) on 4 November 2004.
38. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "An important variation".
39. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Algorithm U".
40. Moffat & Turpin 2002, p. 33.
41. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Interpolation search".
42. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Exercise 22".
43. Perl, Yehoshua; Itai, Alon; Avni, Haim (1978). "Interpolation search—a log log *n* search". *Communications of the ACM*. **21** (7): 550–553. doi:10.1145/359545.359557 (https://doi.org/10.1145/359545.359557).
44. Chazelle, Bernard; Liu, Ding (6 July 2001). *Lower bounds for intersection searching and fractional cascading in higher dimension* (https://dl.acm.org/citation.cfm?doid=380752.380818). 33rd ACM Symposium on Theory of Computing. ACM. pp. 322–329. doi:10.1145/380752.380818 (https://doi.org/10.1145/380752.380818). ISBN 978-1-58113-349-3. Retrieved 30 June 2018.
45. Chazelle, Bernard; Liu, Ding (1 March 2004). "Lower bounds for intersection searching and fractional cascading in higher dimension" (http://www.cs.princeton.edu/~chazelle/pubs/FClowerbounds.pdf) (PDF). *Journal of Computer and System Sciences*. **68** (2): 269–284. doi:10.1016/j.jcss.2003.07.003 (https://doi.org/10.1016/j.jcss.2003.07.003). ISSN 0022-0000 (https://www.worldcat.org/issn/0022-0000). Retrieved 30 June 2018.
46. Ben-Or, Michael; Hassidim, Avinatan (2008). "The Bayesian learner is optimal for noisy binary search (and pretty good for quantum as well)" (http://www2.lns.mit.edu/~avinatan/research/search-full.pdf) (PDF). *49th Symposium on Foundations of Computer Science*. pp. 221–230. doi:10.1109/FOCS.2008.58 (https://doi.org/10.1109/FOCS.2008.58). ISBN 978-0-7695-3436-7.
47. Pelc, Andrzej (2002). "Searching games with errors—fifty years of coping with liars" (http://www.sciencedirect.com/science/article/pii/S0304397501003036). *Theoretical Computer Science*. **270** (1–2): 71–109. doi:10.1016/S0304-3975(01)00303-6 (https://doi.org/10.1016/S0304-3975%2801%2900303-6).
48. Rényi, Alfréd (1961). "On a problem in information theory". *Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei* (in Hungarian). **6**: 505–516. MR 0143666 (https://www.ams.org/mathscinet-getitem?mr=0143666).
49. Pelc, Andrzej (1989). "Searching with known error probability" (http://www.sciencedirect.com/science/article/pii/0304397589900777). *Theoretical Computer Science*. **63** (2): 185–202. doi:10.1016/0304-3975(89)90077-7 (https://doi.org/10.1016/0304-3975%2889%2990077-7).
50. Rivest, Ronald L.; Meyer, Albert R.; Kleitman, Daniel J.; Winklmann, K. *Coping with errors in binary search procedures*. 10th ACM Symposium on Theory of Computing. doi:10.1145/800133.804351 (https://doi.org/10.1145/800133.804351).
51. Høyer, Peter; Neerbek, Jan; Shi, Yaoyun (2002). "Quantum complexities of ordered searching, sorting, and element distinctness". *Algorithmica*. **34** (4): 429–448. arXiv:quant-ph/0102078 (https://arxiv.org/abs/quant-ph/0102078). doi:10.1007/s00453-002-0976-3 (https://doi.org/10.1007/s00453-002-0976-3).
52. Childs, Andrew M.; Landahl, Andrew J.; Parrilo, Pablo A. (2007). "Quantum algorithms for the ordered search problem via semidefinite programming". *Physical Review A*. **75** (3). 032335. arXiv:quant-ph/0608161 (https://arxiv.org/abs/quant-ph/0608161). Bibcode:2007PhRvA..75c2335C (http://adsabs.harvard.edu/abs/2007PhRvA..75c2335C). doi:10.1103/PhysRevA.75.032335 (https://doi.org/10.1103/PhysRevA.75.032335).
53. Grover, Lov K. *A fast quantum mechanical algorithm for database search*. 28th ACM Symposium on Theory of Computing. Philadelphia, PA. pp. 212–219. arXiv:quant-ph/9605043 (https://arxiv.org/abs/quant-ph/9605043). doi:10.1145/237814.237866 (https://doi.org/10.1145/237814.237866).

54. Peterson, William Wesley (1957). "Addressing for random-access storage". *IBM Journal of Research and Development*. 1 (2): 130–146. doi:10.1147/rd.12.0130 (https://doi.org/10.1147/rd.12.0130).
55. "2^n-1". OEIS A000225 (http://oeis.org/A000225). Retrieved 7 May 2016.
56. Lehmer, Derrick (1960). *Teaching combinatorial tricks to a computer. Proceedings of Symposia in Applied Mathematics*. 10. pp. 180–181. doi:10.1090/psapm/010 (https://doi.org/10.1090/psapm/010).
57. Chazelle, Bernard; Guibas, Leonidas J. (1986). "Fractional cascading: I. A data structuring technique" (http://www.cs.princeton.edu/~chazelle/pubs/FractionalCascading1.pdf) (PDF). *Algorithmica*. 1 (1): 133–162. doi:10.1007/BF01840440 (https://doi.org/10.1007/BF01840440).
58. Chazelle, Bernard; Guibas, Leonidas J. (1986), "Fractional cascading: II. Applications" (http://www.cs.princeton.edu/~chazelle/pubs/FractionalCascading2.pdf) (PDF), *Algorithmica*, 1 (1): 163–191, doi:10.1007/BF01840441 (https://doi.org/10.1007/BF01840441)
59. Bentley 2000, §4.1 ("The Challenge of Binary Search").
60. Pattis, Richard E. (1988). "Textbook errors in binary searching". *SIGCSE Bulletin*. 20: 190–194. doi:10.1145/52965.53012 (https://doi.org/10.1145/52965.53012).
61. Bloch, Joshua (2 June 2006). "Extra, extra – read all about it: nearly all binary searches and mergesorts are broken" (http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html). *Google Research Blog*. Retrieved 21 April 2016.
62. Ruggieri, Salvatore (2003). "On computing the semi-sum of two integers" (http://www.di.unipi.it/~ruggieri/Papers/semisum.pdf) (PDF). *Information Processing Letters*. 87 (2): 67–71. doi:10.1016/S0020-0190(03)00263-1 (https://doi.org/10.1016/S0020-0190%2803%2900263-1).
63. Bentley 2000, §4.4 ("Principles").
64. "bsearch – binary search a sorted table" (http://pubs.opengroup.org/onlinepubs/9699919799/functions/bsearch.html). *The Open Group Base Specifications* (7th ed.). The Open Group. 2013. Retrieved 28 March 2016.
65. Stroustrup 2013, p. 945.
66. Unisys (2012), *COBOL ANSI-85 Programming Reference Manual*, 1, pp. 598–601
67. "Package `sort`" (http://golang.org/pkg/sort/). *The Go Programming Language*. Retrieved 28 April 2016.
68. "`java.util.Arrays`" (https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html). *Java Platform Standard Edition 8 Documentation*. Oracle Corporation. Retrieved 1 May 2016.
69. "`java.util.Collections`" (https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html). *Java Platform Standard Edition 8 Documentation*. Oracle Corporation. Retrieved 1 May 2016.
70. "List<T>.BinarySearch method (T)" (https://msdn.microsoft.com/en-us/library/w4e7fxsh%28v=vs.110%29.aspx). *Microsoft Developer Network*. Retrieved 10 April 2016.
71. "`NSArray`" (https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/index.html#//apple_ref/occ/instm/NSArray/indexOfObject:inSortedRange:options:usingComparator:). *Mac Developer Library*. Apple Inc. Retrieved 1 May 2016.
72. "`CFArray`" (https://developer.apple.com/library/mac/documentation/CoreFoundation/Reference/CFArrayRef/index.html#//apple_ref/c/func/CFArrayBSearchValues). *Mac Developer Library*. Apple Inc. Retrieved 1 May 2016.
73. "8.6. bisect — Array bisection algorithm" (https://docs.python.org/3.6/library/bisect.html#module-bisect). *The Python Standard Library*. Python Software Foundation. Retrieved 26 March 2018.
74. Fitzgerald 2007, p. 152.

Works

- Bentley, Jon (2000). *Programming pearls* (2nd ed.). Addison-Wesley. ISBN 978-0-201-65788-3.
- Butterfield, Andrew; Ngondi, Gerard E. (2016). *A dictionary of computer science* (7th ed.). Oxford, UK: Oxford University Press. ISBN 978-0-19-968897-5.
- Chang, Shi-Kuo (2003). *Data structures and algorithms*. *Software Engineering and Knowledge Engineering*. **13**. Singapore: World Scientific. ISBN 978-981-238-348-8.

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). *Introduction to algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 978-0-262-03384-8.
- Fitzgerald, Michael (2007). *Ruby pocket reference*. Sebastopol, California: O'Reilly Media. ISBN 978-1-4919-2601-7.
- Goldman, Sally A.; Goldman, Kenneth J. (2008). *A practical guide to data structures and algorithms using Java*. Boca Raton, Florida: CRC Press. ISBN 978-1-58488-455-2.
- Kasahara, Masahiro; Morishita, Shinichi (2006). *Large-scale genome sequence processing*. London, UK: Imperial College Press. ISBN 978-1-86094-635-6.
- Knuth, Donald (1997). *Fundamental algorithms*. The Art of Computer Programming. **1** (3rd ed.). Reading, MA: Addison-Wesley Professional. ISBN 978-0-201-89683-1.
- Knuth, Donald (1998). *Sorting and searching*. The Art of Computer Programming. **3** (2nd ed.). Reading, MA: Addison-Wesley Professional. ISBN 978-0-201-89685-5.
- Knuth, Donald (2011). *Combinatorial algorithms*. The Art of Computer Programming. **4A** (1st ed.). Reading, MA: Addison-Wesley Professional. ISBN 978-0-201-03804-0.
- Moffat, Alistair; Turpin, Andrew (2002). *Compression and coding algorithms*. Hamburg, Germany: Kluwer Academic Publishers. doi:10.1007/978-1-4615-0935-6 (https://doi.org/10.1007/978-1-4615-0935-6). ISBN 978-0-7923-7668-2.
- Sedgewick, Robert; Wayne, Kevin (2011). *Algorithms* (http://algs4.cs.princeton.edu/home/) (4th ed.). Upper Saddle River, New Jersey: Addison-Wesley Professional. ISBN 978-0-321-57351-3. Condensed web version: ; book version .
- Stroustrup, Bjarne (2013). *The C++ programming language* (4th ed.). Upper Saddle River, New Jersey: Addison-Wesley Professional. ISBN 978-0-321-56384-2.

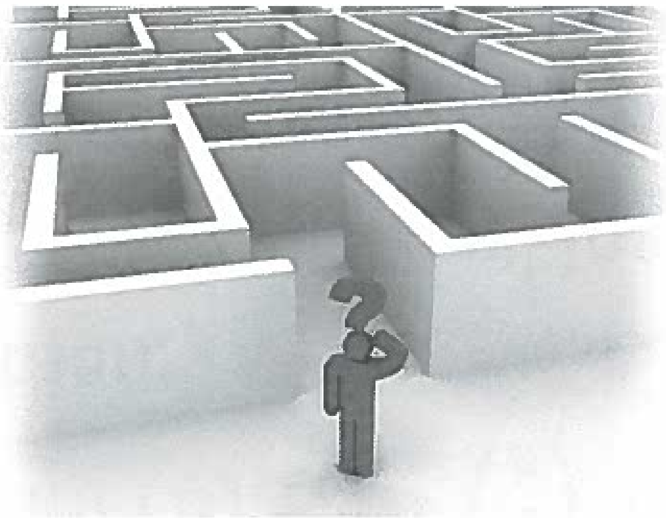
External links

- NIST Dictionary of Algorithms and Data Structures: binary search (https://xlinux.nist.gov/dads/HTML/binarySearch.html)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Binary_search_algorithm&oldid=850911217"

This page was last edited on 18 July 2018, at 19:51 (UTC).

July 21, 2018 20



Master Method

Examples

Design and Analysis
of Algorithms I

The Master Method

no. of recursive calls.

- If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$ work outside the recursive calls.

factor sub problem size is smaller than the original problem size.

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Example #1

Merge Sort

$2^1 = 2$

$$\left.\begin{aligned} a &= 2 \\ b &= 2 \\ d &= 1 \end{aligned}\right\} \quad b^d = \boxed{a} \Rightarrow \quad Case\ 1$$

merge

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

Tim Roughgarden

FV 2

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

$1 = 2^0$

- 1, 2, 0 [Case 1]
- 1, 2, 1 [Case 2]
- 2, 2, 0 [Case 3]
- 2, 2, 1 [Case 1]

$$a = b^d => T(n) = O(n^d \log n) = \boxed{O(\ \log n)}$$

$\overline{a}\ \overline{b} \times \overline{c}\ \overline{d}$

4 multiplications: ac ad bc bd

Example #3

Integer Multiplication Algorithm # 1 (w/o Gauss trick)

- a = 4
- b = 2 (divide by 2)
- d = 1

$$\underbrace{a, b, d}\ \ b^d = 2 < a \ \ (Case\ \ 3)$$

$$=> T(n) = O(n^{\log_b a}) = O(n^{\overbrace{\log_2 4}^{=2}})$$

$$= O(n^2)$$

adding and multiplication $O(n^1)$.

Same as grade-school algorithm

no improvement in comparison to the usual multiplication

Tim Roughgarden

EX 3

Ter
ver

Where are the respective values of a, b, d for Gauss's recursive integer multiplication algorithm, and which case of the Master Method does this correspond to?

- ○ 2, 2, 1 [Case 1]
- ○ 3, 2, 1 [Case 1]
- ○ 3, 2, 1 [Case 2]
- ○ 3, 2, 1 [Case 3]

Better than the grade-school algorithm!!!

$$a = 3,\ b^d = 2\ a > b^d \quad (Case\ 3)$$

$$=> T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Example #5

Strassen's Matrix Multiplication Algorithm

matrix of size $n \longrightarrow$ matrix of size $\frac{n}{2}$.

a = 7 recursive calls:

b = 2

d = 2

why

$$b^d = 4 < a \quad (Case\ 3)$$

$$=> T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

=> beats the naïve iterative algorithm !

Tim Roughgarden

Example #6

Fictitious Recurrence

$$T(n) \leq 2T(n/2) + O(n^2)$$

$\Rightarrow$ a = 2

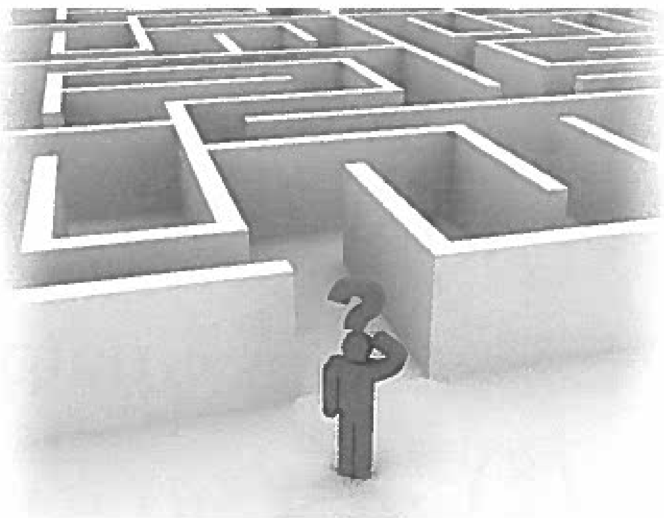
$\Rightarrow$ b = 2

$\Rightarrow$ d = 2

$$b^d = 4 > a \quad (Case\ 2)$$

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

21



Design and Analysis of Algorithms I

Master Method Proof (Part I)

The concept each case is more important than the proof itself.

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Preamble

Assume : recurrence is

I. $T(1) \leq c$ (For some constant c)

II. $T(n) \leq aT(n/b) + cn^d$ → from big-oh notation

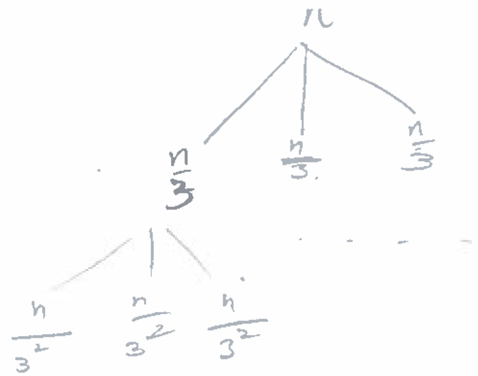
III. And n is a power of b.

(general case is similar, but more tedious)

Idea : generalize MergeSort analysis.

(i.e., use a recursion tree)

Tim Roughgarden



What is the pattern ? Fill in the blanks in the following statement: at each level $j = 0,1,2,\ldots,\log_b n$, there are <blank> subproblems, each of size <blank>

# of times you can divide n by b before reaching 1

- ○ a^j and n/a^j, respectively.
- ○ a^j and n/b^j, respectively.
- ○ b^j and n/a^j, respectively.
- ○ b^j and n/b^j, respectively.

a^j subproblems. of size $\frac{n}{b^j}$.

The Recursion Tree

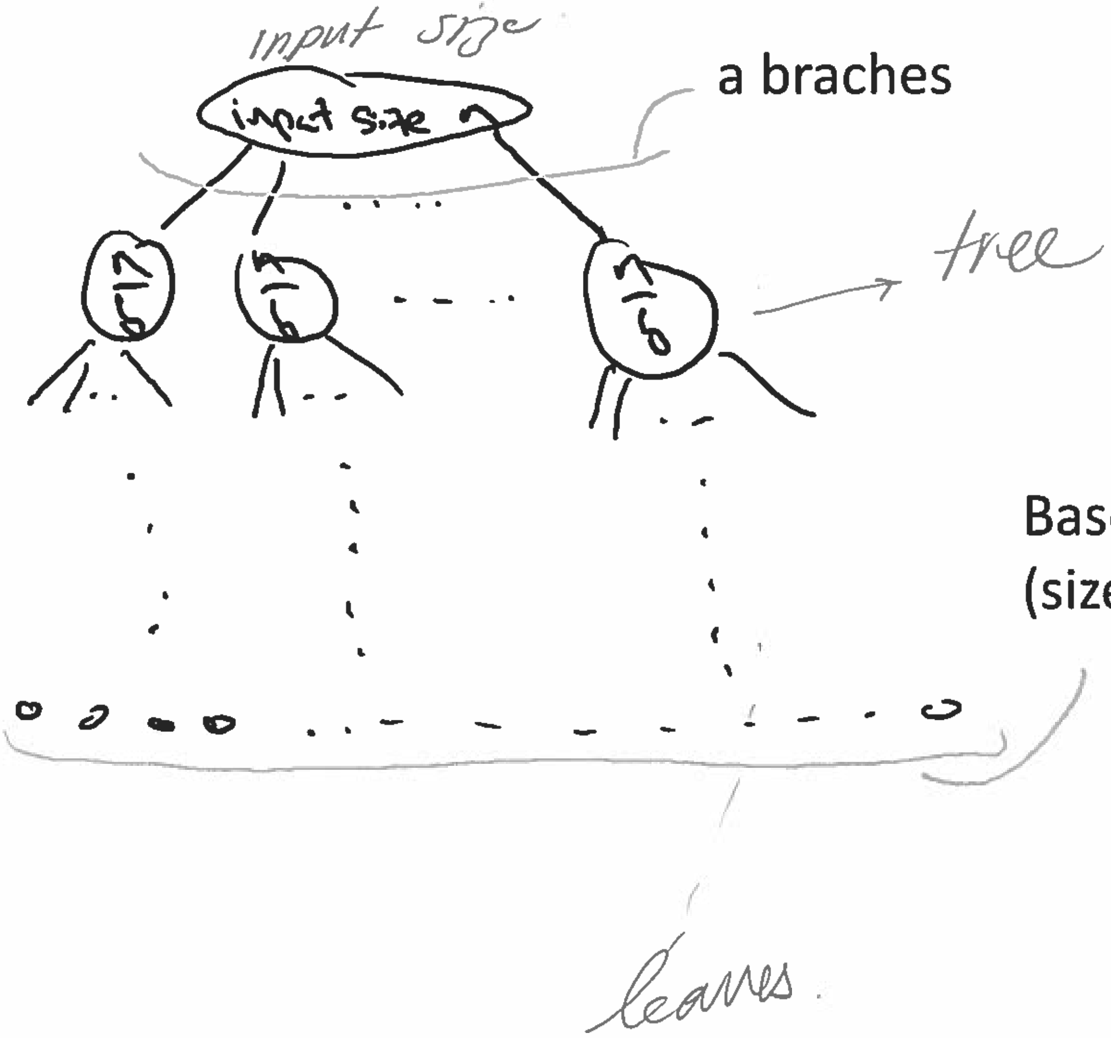
Level 0

Level 1

.
.
.
.

Level $\log_b n$

a braches



Base cases
(size 1)

Tim Roughgarden

Proof I-3

Work at a Single Level

Total work at level j [ignoring work in recursive calls]

$$\leq \underbrace{a^j}_{\text{\# of level-j subproblems}} \cdot c \cdot \left(\frac{n}{b^j}\right)^d = cn^d \cdot \left(\frac{a}{b^d}\right)^j$$

Work per level-j subproblem

$O(n^d) = cn^d$

Size of each level-j subproblem

Tim Roughgarden

Total Work

<u>Summing over all levels j = 0,1,2,..., $\log_b n$:</u>

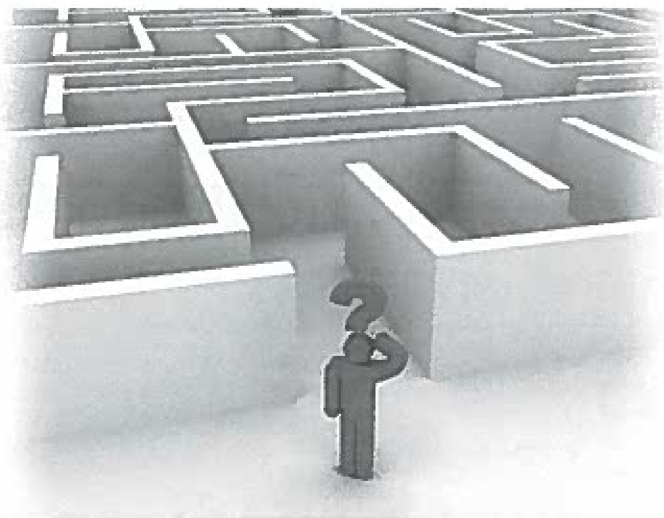
// total # of levels.

$$\text{Total work} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \qquad (*)$$

Tim Roughgarden

Proof I – 4

July 21, 2018 22



Master Method Intuition for the 3 Cases

Design and Analysis of Algorithms I

How To Think About (*)

- Our upper bound on the work at level j:

$$\boxed{cn^d \times \left(\frac{a}{b^d}\right)^j}$$

The cases are actually saying how a changes with d.

Interpretation

a = rate of subproblem proliferation (RSP)

evil: bcs as we go down the tree there are more subproblems.

b^d = rate of work shrinkage (RWS)

(per subproblem)

good — doing less work for each problem.

$a > b^d$ not good

$a < b^d$ good

Tim Roughgarden

Which of the following statements are true?
(Check all that apply.)

- ☐ If RSP < RWS, then the amount of work is decreasing with the recursion level j. $\frac{a}{b^d} < 1 \Rightarrow \left(\frac{a}{b^d}\right)^j$ gets smaller decreasing.
- ☐ If RSP > RWS, then the amount of work is increasing with the recursion level j.
- ☐ No conclusions can be drawn about how the amount of work varies with the recursion level j unless RSP and RWS are equal.
- ☐ If RSP and RWS are equal, then the amount of work is the same at every recursion level j.

$\frac{a}{b^d} = 1 \Rightarrow \left(\frac{a}{b^d}\right)^j$ remains the same.

Te
blc

Or
bu
13
24

In fact the three case are: (1) work done per level is the same. (2) work is decreasing with the level (2 with) (3) work is increasing in the level (leaves are more)

Intuition for the 3 Cases

Upper bound for level j: $cn^d \times (\frac{a}{b^d})^j$

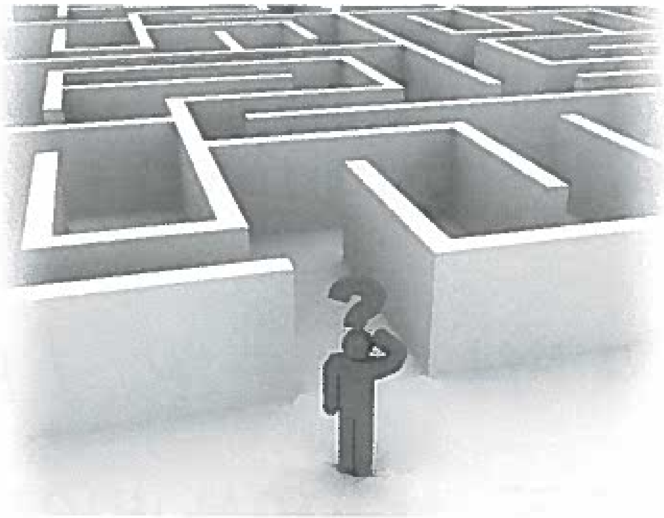
$\to (\frac{a}{b^d})^j = 1 \Rightarrow$ work in each level $= O(n^d)$. We have $\log_b n$ levels thus run time is $O(n \log n)$

1. RSP = RWS => Same amount of work each level (like Merge Sort) [expect $O(n^d \log(n))$]
2. RSP < RWS => less work each level => most work at the root $(\frac{a}{b^d})^j \to 0 \Rightarrow O(n^d)$ root level [might expect $O(n^d)$]
3. RSP > RWS => more work each level => most work at the leaves [might expect O(# leaves)]

$O(n^{\log_b a})$

Tim Roughgarden

July 21, 2018 23



Master Method Proof (Part II)

Design and Analysis
of Algorithms I

Proof II — 1

The Story So Far/Case 1

= 1 for all j

Total work: $\leq cn^d \times \underbrace{\sum_{j=0}^{\log_b n} \boxed{\left(\frac{a}{b^d}\right)^j}}_{= (\log_b n + 1)}$ (*)

=1

case I: $If \quad a = b^d, \ then$

$$(*) = cn^d(\log_b n + 1)$$

$$= O(n^d \log n)$$

[end Case 1]

Tim Roughgarden

geometric series

Basic Sums Fact

For $r \neq 1$, we have

$$1 + r + r^2 + r^3 + \ldots + r^k = \frac{r^{k+1} - 1}{r - 1}$$

Proof : by induction (you check)

Upshot:

Independent of k

in this case the very first term dominates (1)

1. If r<1 is constant, RHS is <= $\frac{1}{1-r}$ = a constant
   I.e., 1st term of sum dominates
2. If r>1 is constant, RHS is <= $r^k \cdot \left(1 + \frac{1}{r-1}\right)$
   I.e., last term of sum dominates

at most, let's say $2r^k$.

Tim Roughgarden

Proof II_2

Case 2

Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$ (*)

:= r

<= a constant
(independent of n)
[by basic sums fact]

$If \quad a < b^d \quad [RSP < RWS]$

$= O(n^d)$

[total work dominated by top level]

Tim Roughgarden

Case 3

Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$ (*)

:= r > 1

<= constant * largest term

$If \quad a > b^d \quad [RSP > RWS]$

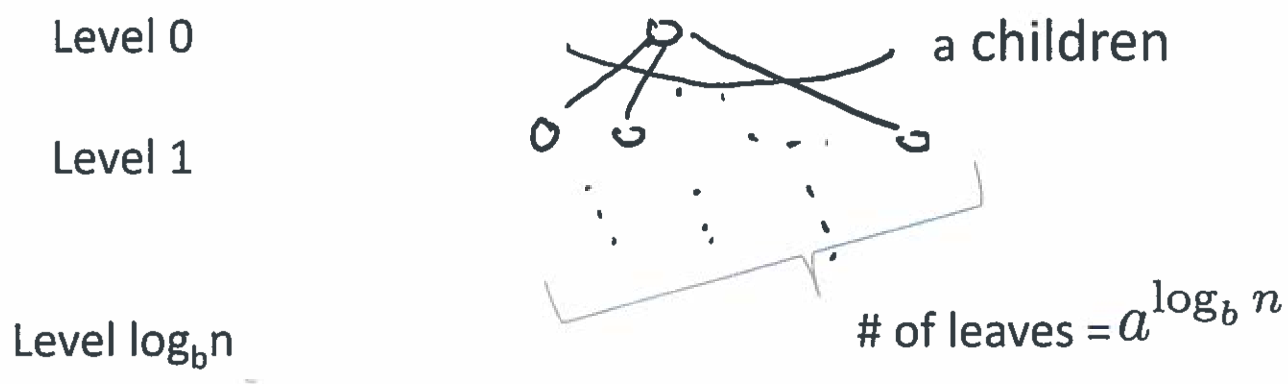
$(*) = O(n^d \cdot (\frac{a}{b^d})^{\log_b n})$

largest term of sum

$Note \; : \quad b^{-d\log_b n} = (b^{\log_b n})^{-d} = n^{-d}$

$So \; : \quad (*) = O(a^{\log_b n})$

Tim Roughgarden



Which of the following quantities is equal to $a^{\log_b n}$?

- ○ The number of levels of the recursion tree.
- ○ The number of nodes of the recursion tree.
- ○ The number of edges of the recursion tree.
- ○ The number of leaves of the recursion tree.

Te
ve

Case 3 continued

Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$ (*)

$So: \quad (*) = O(a^{\log_b n}) = O(\#\ leaves)$

$Note: \quad a^{\log_b n} = n^{\log_b a}$

More intuitive

Simpler to apply

$[Since \quad (\log_b n)(\log_b a) = (\log_b a)(\log_b n)]$

[End Case 3]

$a = b^{\lg_b^a} \Rightarrow a^{\lg_b^n} = b^{\lg_b^a \cdot \lg_b^n} = b^{\lg_b^n \cdot \lg_b^a} = \boxed{n^{\lg_b^a}}$

Tim Roughgarden

Proof#_4

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Tim Roughgarden