3. Searching

The Search Problem

Provided

- A set of data sets

examples
telephone book, dictionary, symbol table

- Each dataset has a key k.
- Keys are comparable: unique answer to the question $k_1 \leq k_2$ for keys k_1, k_2.

Task: find data set by key k.

Search in Array

Provided

- Array A with n elements $(A[1], \ldots, A[n])$.
- Key b

Wanted: index k, $1 \leq k \leq n$ with $A[k] = b$ or "not found".

1	2	3	4	5	6	7	8	9	10
22	20	32	10	35	24	42	38	28	41

Linear Search

Traverse the array from $A[1]$ to $A[n]$.

- *Best case:* 1 comparison.
- *Worst case:* n comparisons.
- Assumption: each permutation of the n keys with same probability. *Expected* number of comparisons:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}.$$

Search in a Sorted Array

Provided

- Sorted array A with n elements $(A[1], \dots, A[n])$ with $A[1] \leq A[2] \leq \dots \leq A[n]$.
- Key b

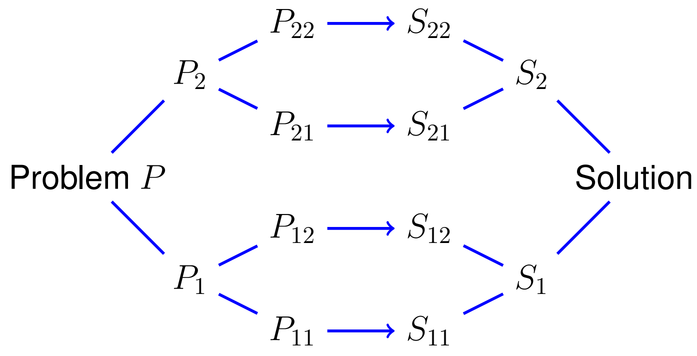
Wanted: index k, $1 \leq k \leq n$ with $A[k] = b$ or "not found".

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

divide et impera

Divide and Conquer

Divide the problem into subproblems that contribute to the simplified computation of the overal problem.



Divide and Conquer!

Search $b = 23$.

10	20	22	24	28	32	35	38	41	42	$b < 28$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	$b > 20$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	$b > 22$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	$b < 24$
1	2	3	4	5	6	7	8	9	10	
10	20	22	24	28	32	35	38	41	42	erfolglos
1	2	3	4	5	6	7	8	9	10	

Binary Search Algorithm BSearch$(A[l..r], b)$

Input : Sorted array A of n keys. Key b. Bounds $1 \leq l \leq r \leq n$ or $l > r$ beliebig.
Output : Index of the found element. 0, if not found.

$m \leftarrow \lfloor (l + r)/2 \rfloor$
if $l > r$ **then** // Unsuccessful search
  return *NotFound*
else if $b = A[m]$ **then** // found
  return m
else if $b < A[m]$ **then** // element to the left
  return BSearch$(A[l..m-1], b)$
else // $b > A[m]$: element to the right
  return BSearch$(A[m+1..r], b)$

Analysis (worst case)

Recurrence ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Compute:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + c \cdot \log_2 n = d + c \cdot \log_2 n \end{aligned}$$

$\Rightarrow$ Assumption: $T(n) = d + c \log_2 n$

Analysis (worst case)

$$T(n) = \begin{cases} d & \text{if } n = 1, \\ T(n/2) + c & \text{if } n > 1. \end{cases}$$

Guess : $T(n) = d + c \cdot \log_2 n$

Proof by induction:

- Base clause: $T(1) = d$.
- Hypothesis: $T(n/2) = d + c \cdot \log_2 n/2$
- Step: $(n/2 \to n)$

$$T(n) = T(n/2) + c = d + c \cdot (\log_2 n - 1) + c = d + c \log_2 n.$$

Result

Theorem

The binary sorted search algorithm requires $\Theta(\log n)$ fundamental operations.

Iterative Binary Search Algorithm

Input : Sorted array A of n keys. Key b.
Output : Index of the found element. 0, if unsuccessful.

```
l ← 1; r ← n
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if A[m] = b then
        return m
    else if A[m] < b then
        l ← m + 1
    else
        r ← m − 1
return NotFound;
```

Correctness

Algorithm terminates only if A is empty or b is found.

Invariant: If b is in A then b is in domain $A[l..r]$

Proof by induction

- Base clause $b \in A[1..n]$ (oder nicht)
- Hypothesis: invariant holds after i steps.
- Step:
  $b < A[m] \Rightarrow b \in A[l..m-1]$
  $b > A[m] \Rightarrow b \in A[m+1..r]$

4. Selection

Min and Max

? To separately find minimum an maximum in $(A[1], \dots, A[n])$, $2n$ comparisons are required. (How) can an algorithm with less than $2n$ comparisons for both values at a time can be found?

! Possible with $\frac{3}{2}n$ comparisons: compare 2 elemetns each and then the smaller one with min and the greater one with max.

The Problem of Selection

Input

- unsorted array $A = (A_1, \dots, A_n)$ with pairwise different values
- Number $1 \leq k \leq n$.

Output $A[i]$ with $|\{j : A[j] < A[i]\}| = k - 1$

Special cases

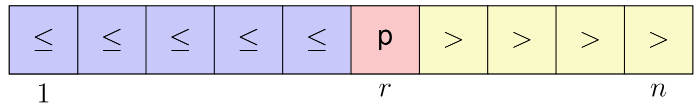
$k = 1$: Minimum: Algorithm with n comparison operations trivial.
$k = n$: Maximum: Algorithm with n comparison operations trivial.
$k = \lfloor n/2 \rfloor$: Median.

Approaches

- Repeatedly find and remove the minimum $\mathcal{O}(k \cdot n)$.
  Median: $\mathcal{O}(n^2)$
- Sorting (covered soon): $\mathcal{O}(n \log n)$
- Use a pivot $\mathcal{O}(n)$!

Use a pivot

1. Choose a *pivot* p
2. Partition A in two parts, thereby determining the rank of p.
3. Recursion on the relevant part. If $k = r$ then found.



Algorithmus Partition($A[l..r], p$)

Input : Array A, that contains the pivot p in the interval $[l, r]$ at least once.
Output : Array A partitioned in $[l..r]$ around p. Returns position of p.

while $l \leq r$ **do**
    while $A[l] < p$ **do**
        $l \leftarrow l + 1$
    while $A[r] > p$ **do**
        $r \leftarrow r - 1$
    swap($A[l]$, $A[r]$)
    if $A[l] = A[r]$ **then**
        $l \leftarrow l + 1$

return l-1

Correctness: Invariant

Invariant I: $A_i \leq p \; \forall i \in [0, l)$, $A_i \geq p \; \forall i \in (r, n]$, $\exists k \in [l, r] : A_k = p$.

while $l \leq r$ **do**
    — I
    while $A[l] < p$ **do**
        $l \leftarrow l + 1$
    — I und $A[l] \geq p$
    while $A[r] > p$ **do**
        $r \leftarrow r - 1$
    — I und $A[r] \leq p$
    swap($A[l]$, $A[r]$)
    — I und $A[l] \leq p \leq A[r]$
    if $A[l] = A[r]$ **then**
        $l \leftarrow l + 1$
    — I

return l-1

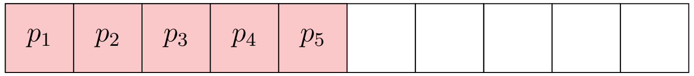
Correctness: progress

while $l \leq r$ **do**
- **while** $A[l] < p$ **do** — progress if $A[l] < p$
  - $l \leftarrow l + 1$
- **while** $A[r] > p$ **do** — progress if $A[r] > p$
  - $r \leftarrow r - 1$
- swap($A[l]$, $A[r]$) — progress if $A[l] > p$ oder $A[r] < p$
- **if** $A[l] = A[r]$ **then** — progress if $A[l] = A[r] = p$
  - $l \leftarrow l + 1$

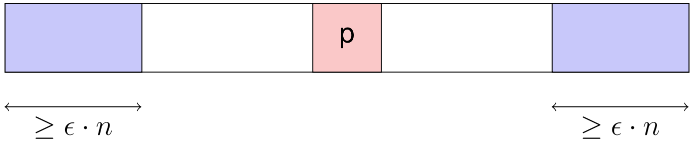
return l-1

Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(n^2)$



A good pivot has a linear number of elements on both sides.



Analysis

Partitioning with factor q $(0 < q < 1)$: two groups with $q \cdot n$ and $(1 - q) \cdot n$ elements (without loss of generality $g \geq 1 - q$).

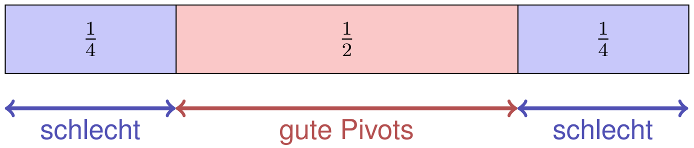
$$T(n) \leq T(q \cdot n) + c \cdot n$$

$$= c \cdot n + q \cdot c \cdot n + T(q^2 \cdot n) = \ldots = c \cdot n \sum_{i=0}^{\log_q(n)-1} q^i + T(1)$$

$$\leq c \cdot n \underbrace{\sum_{i=0}^{\infty} q^i}_{\text{geom. Reihe}} + d = c \cdot n \cdot \frac{1}{1-q} + d = \mathcal{O}(n)$$

How can we achieve this?

Randomness to our rescue (Tony Hoare, 1961). In each step choose a random pivot.



Probability for a good pivot in one trial: $\frac{1}{2} =: \rho$.

Probability for a good pivot after k trials: $(1 - \rho)^{k-1} \cdot \rho$.

Expected value of the geometric distribution: $1/\rho = 2$

Algorithm Quickselect ($A[l..r], k$)

Input : Array A with length n. Indices $1 \leq l \leq k \leq r \leq n$, such that for all $x \in A[l..r]$: $|\{j|A[j] \leq x\}| \geq l$ and $|\{j|A[j] \leq x\}| \leq r$.
Output : Value $x \in A[l..r]$ with $|\{j|A[j] \leq x\}| \geq k$ and $|\{j|x \leq A[j]\}| \geq n-k+1$

```
if l=r then
    return A[l];
x ← RandomPivot(A[l..r])
m ← Partition(A[l..r], x)
if k < m then
    return QuickSelect(A[l..m − 1], k)
else if k > m then
    return QuickSelect(A[m + 1..r], k)
else
    return A[k]
```


Algorithm RandomPivot ($A[l..r]$)

Input : Array A with length n. Indices $1 \leq l \leq i \leq r \leq n$
Output : Random "good" pivot $x \in A[l..r]$

```
repeat
    choose a random pivot x ∈ A[l..r]
    p ← l
    for j = l to r do
        if A[j] ≤ x then p ← p + 1
until ⌊(3l+r)/4⌋ ≤ p ≤ ⌈(l+3r)/4⌉
return x
```

This algorithm is only of theoretical interest and delivers a good pivot in 2 expected iterations. Practically, in algorithm QuickSelect a uniformly chosen random pivot can be chosen or a deterministic one such as the median of three elements.