HUMBOLDT-UNIVERSITÄT · ZU BERLIN ·

Algorithms and Data Structures

Searching in Lists

Ulf Leser

This Course

Introduction	2
Abstract Data Types	1
Complexity analysis	1
Styles of algorithms	1
List implementations	1
Sorting (lists)	3
Searching (in (sorted) lists)	4
Hashing (to manage lists)	2
Trees (to manage lists)	4
Graphs (no lists!)	5
Sum	**~9/25**

Topics of Next Lessons

- Search: Given a (sorted or unsorted) list A with |A|=n elements (integers). Check whether a given value c is contained in A or not
  - Search returns true or false
  - If A is sorted, we can exploit transitivity of "≤" relation
  - Fundamental problem with a zillion applications
- Select: Given an unsorted list A with |A|=n elements (integers). Return the i'th largest element of A.
  - Returns an element of A
  - The sorted case is trivial – return A[i]
  - Interesting problem (especially for median) with some applications
  - [Interesting proof]

Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
- Selecting in Unsorted Lists

Searching in an Unsorted List

- No magic
- Compare c to every element of A
- Worst case ($c \notin A$): O(n)
- Average case ($c \in A$)
  - If c is at position i, we require i tests
  - All positions are equally likely: probability 1/n
  - This gives

```
A: unsorted_int_array;
c: int;
for i := 1.. |A| do
  if A[i]=c then
    return true;
  end if;
end for;
return false;
```

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} * \frac{n^2+n}{2} = \frac{n+1}{2} = O(n)$$

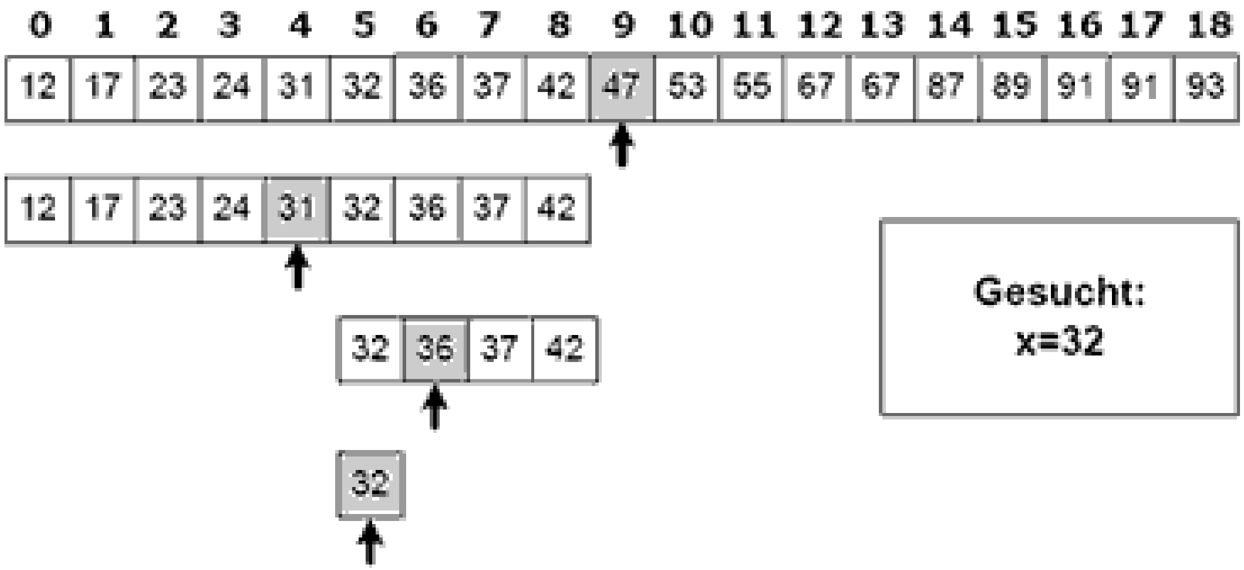
- Sequential access: Same for array, linked lists, …

Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
  - Binary Search
  - Fibonacci Search
  - Interpolation Search
- Selecting in Unsorted Lists

Binary Search (binsearch)

- If A is sorted, we can be much faster
- Binary Search: Exploit transitivity



Source: http://hki.uni-koeln.de

Recursive versus Iterative Binsearch

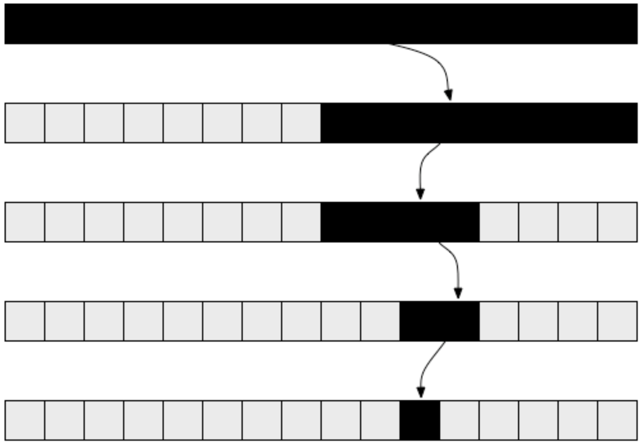
- Recursive binsearch uses only end-recursion
- Equivalent iterative program is more space-efficient
  - We don’t need old values for l,r – no call stack
  - O(1) additional space

```
func bool binsearch(A: sorted_array;
                    c,l,r : int) {
  If l>r then
    return false;
  end if;
  m := l+((r-l) div 2);
  If c<A[m] then
    return binsearch(A, c, l, m-1);
  else if c>A[m] then
    return binsearch(A, c, m+1, r);
  else
    return true;
  end if;
}
```

```
A: sorted_int_array;
c: int;
l := 1;
r := |A|;
while l≤r do
  m := l+(r-l) div 2;
  if c<A[m] then
    r := m-1;
  else if c>A[m] then
    l := m+1;
  else
    return true;
end while,
return false;
```

Complexity of Binsearch

- In every call to binsearch (or every while-loop), we only do constant work
  - Independent of n
- With every call, we reduce the size of sub-array by ~50%
  - We call binsearch once with n, with n/2, with n/4, …
- Binsearch has worst-case complexity O(log(n))
- Average case only marginally better
  - We only stop if we find c before the interval has size 1
  - Chances to "hit" target in the middle of an interval are low in most cases
  - See Ottmann/Widmayer



Source: railspikes.com

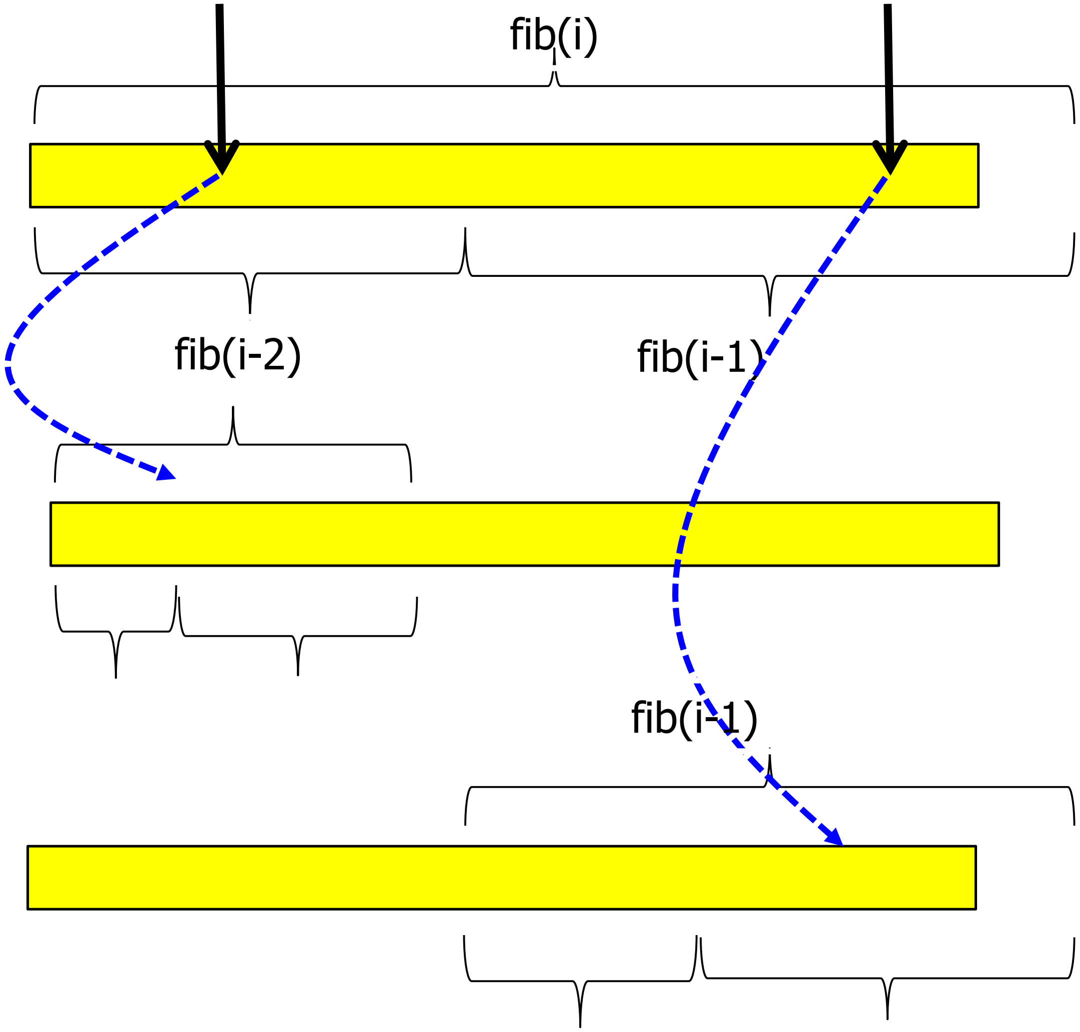
Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
  - Binary Search
  - Fibonacci Search
  - Interpolation Search
- Selecting in Unsorted Lists

Searching without Divisions

- Can we search in O(log(n)) without complex arithmetics?
  - Simple arithmetic operations are faster on real hardware
  - But: Binsearch usually uses bit shift (div 2) – very fast
- Fibonacci search: O(log(n)) without division/multiplication
  - Fibonacci search has slightly better access locality (cache)
  - Interesting: O(log(n)) without the "always 50%" trick
- Recall Fibonacci numbers
  - fib(1)=fib(2)=1; fib(i)=fib(i-1)+fib(i-2)
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, …
  - Observation: fib(i-2) is roughly 1/3, fib(i-1) roughly 2/3 of fib(i)

Fibonacci Search: Idea



- Let fib(i) be the smallest fib-number >|A|
- If A[fib(i-2)]=c: stop
- Otherwise, search in [1 … fib(i-2)] or [fib(i-2)+1 … n]
- Beware out-of-range part A[n+1…fib(i)]
- No divisions

Algorithm (assume |A|=fib(n)-1)

- 3-6: Search at A[fib(i-2)]
  - With fib2, fib3 we can compute all other fib's
  - fib(i)=fib(i-1)+fib(i-2)
  - fib(i-1)=fib(i-2)+fib(i-3)
  - …
- 7-24: Partition A at descending Fibonacci numbers
- After each comparison, update fib3 and fib2

```
1.  A: sorted_int_array;
2.  c: int;
3.  compute i;   #fib(i) smallest …
4.  fib3 := fib(i-3); # Precomputed
5.  fib2 := fib(i-2); # Precomputed
6.  m := fib2;
7.  repeat
8.    if c>A[m] then
9.      if fib3=0 then return false
10.     else
11.       m := m+fib3;
12.       tmp := fib3;
13.       fib3 := fib2-fib3;
14.       fib2 := tmp;
15.     end if;
16.   else if c<A[m]
17.     if fib2=1 then return false
18.     else
19.       m := m-fib3;
20.       fib2 := fib2 – fib3;
21.       fib3 := fib3 – fib2;
22.     end if;
23.   else return true;
24. until true;
```

Example (recall: 1,1,2,3,5,…)

Search 3 in
{1,2,3};
i=4

fib2	fib3	m
2	1	2
1	1	3

true

Search 6 in
{1,2,3,4};
i=5

fib2	fib3	m
3	2	3
2	1	4
1	0	

false

Search 100 in
{1…10000}

fib2	fib3	m
4181	2584	4181
1597	987	1597
…	…	…

```
A: sorted_int_array;
c: int;
compute i;   #fib(i) smallest …
fib3 := fib(i-3);
fib2 := fib(i-2);
m := fib2;
repeat
  if c>A[m] then
    if fib3=0 then return false
    else
      m := m+fib3;
      tmp := fib3;
      fib3 := fib2-fib3;
      fib2 := tmp;
    end if;
  else if c<A[m]
    if fib2=1 then return false
    else
      m := m-fib3;
      fib2 := fib2 – fib3;
      fib3 := fib3 – fib2;
    end if;
  else return true;
until true;
```

Complexity

- Worst-case: c is always in the larger fraction of A
  - We roughly call once for n, once for 2n/3, once for 4n/9, …
- Formula of Moivre-Binet: For large i …

$$fib(i) \sim \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{i} \right] \sim k * 1.62^{i}$$

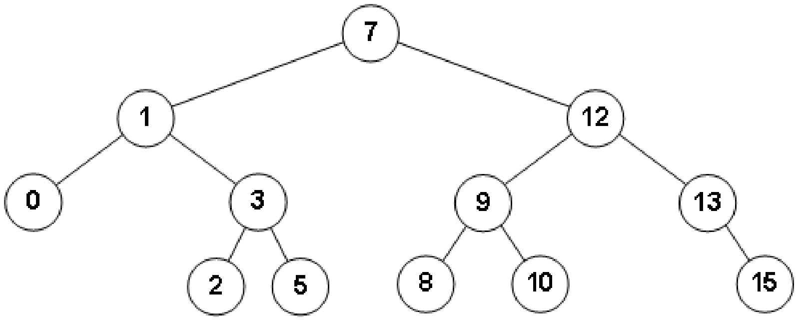
- We find i such that $fib(i-1) \leq n \leq fib(i) \sim k*1{,}62^{i}$
- In worst-case, we make ~i comparisons
  - We break the array i times
- Since $i=\log_{1,62}(n/k)$, we are in O(log(n))

Main message

- If you break an array always in the middle, you can do this at most O(log(n)) times
- If you break an array always at 1/3 and 2/3, you also can do this at most O(log(n)) times
- What if we break an array always at 1/10 – 9/10?
  - Wait a minute

Searching without Math (sketch – details later)

- We actually can solve the search problem in O(log(n)) using only comparisons (no additions etc.)
- Transform A into a balanced binary search tree
  - At every node, the depth of the two subtrees differ by at most 1
  - At every node n, all values in the left (right) subtree are smaller (larger) than n
- Search
  - Recursively compare c to node labels and descend left/right
  - Balanced bin-tree has depth O(log(n))
  - We need at most log(n) comparisons – and nothing else



Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
  - Binary Search
  - Fibonacci Search
  - Interpolation Search
- Selecting in Unsorted Lists

Interpolation Search

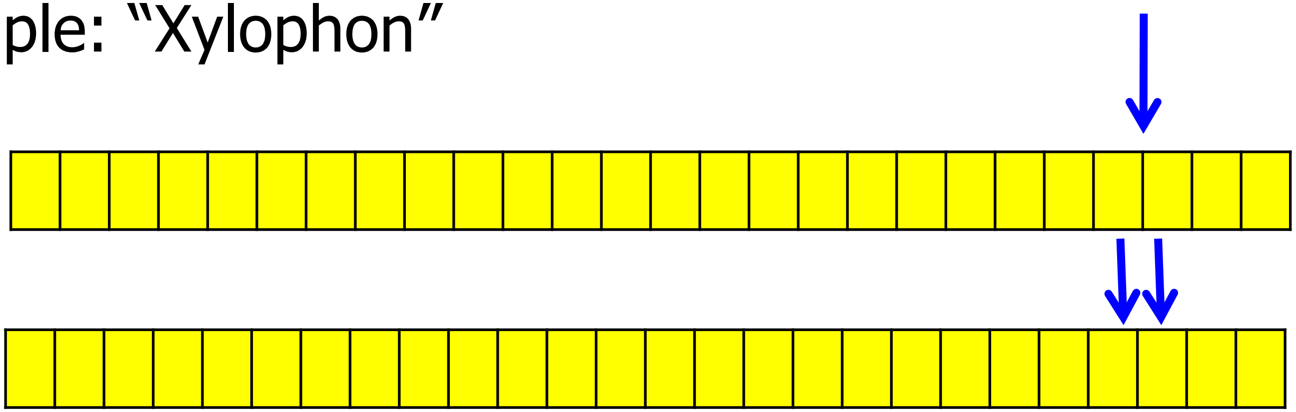
- Imagine you have a telephone book and search for „Zacharias"
- Will you open the book in the middle?
- We can exploit additional knowledge about the keys
- Interpolation Search: Estimate where c lies in A based on the distribution of values in A
  - Simple: Use max and min values in A and assume equal distribution
  - Complex: Approximation of real distribution (histograms, …)

Simple Interpolation Search

- Assume equal distribution – values within A are equally distributed in range [A[1], A[n]]
- Best guess for the rank (position in A) of c

$$rank(c) = l + (r-l) * \frac{c - A[l]}{A[r] - A[l]}$$

- Idea: Use m=rank(c) and proceed recursively
- Example: “Xylophon”



Analysis

- On average, Interpolation Search on equally distributed data requires O(log(log(n)) comparison
  - Proof: See [OW94]
- But: Worst-case is O(n)
  - If concrete distribution deviates heavily from expected distribution
  - E.g., A contains "aaa" and all other names>"Xanthippe"
- Further disadvantage: In each phase, we perform ~4 adds/subs and 2*mults/divs
  - Assume this takes 12 cycles (1 mult/div = 4 cycles)
  - Binsearch requires 2*adds/subs + 1*shift ~3 cycles
  - Even for $n=2^{32}$~4E9, this yields 12*log(log(4E9))~72 ops versus 3*log(4E9)~90 ops – not that much difference

Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
- Selecting in Unsorted Lists
  - Naïve or clever

Quantiles

- Recall: The median of a list is its middle value
  - Sort all values and take the one in the middle
- Generalization: x%-quantiles
  - Sort all values and take the value at x% of all values
  - Typical: 25, 75, 90, -quantiles
    - How long do 90% of all students need to obtain their degree?
  - The 25%, 50%, 75% are called quartiles
  - Median = 50%-quantile

Selection Problem

- Definition
  The selection problem is to find the x%-quantile of a set A of unsorted values
- Solutions
  - We can sort A and then access the quantile directly
  - Thus, O(n*log(n)) is easy
  - It is easy to see that we have to look at least at each value once; thus, the problem is in Ω(n)
  - Can we solve this problem in linear time?

Observation and Example: Top-k Problem

- Top-k: Find the k largest values in A
- For constant k, a naïve solution is linear (and optimal)
  - repeat k times
  -   go through A and find largest value v;
  -   remove v from A;
  - return v
  - Requires k*|A|=O(|A|) comparisons
- But if k=c*|A|, we are in O(c*|A|*|A|)=O(|A|^2)
  - For any constant factor c
  - We measure complexity in size of the input
  - It is decisive whether c is part of the input or not

Selection Problem in Linear Time

- We sketch an algorithm which solves the selection problem in linear time
  - Actually, we solve the equivalent problem of returning the k'th value in the sorted A (without sorting A)
- Interesting from a theoretical point-of-view
- Practically, the algorithm is of no importance because the linear factor gets enormously large
- It is instructive to see why (and where)

Algorithm

- Recall QuickSort: Chose pivot element p, divide array wrt p, recursively sort both partitions using the same trick
- We reuse the idea: Chose pivot element p, divide array wrt p, recursively select in the one partition that must contain the k'th element

```
func integer divide(A array;
                    l,r integer) {
  …
  while true
    repeat
      i := i+1;
    until A[i]>=val;
    repeat
      j := j-1;
    until A[j]<=val or j<i;
    if i>j then
      break while;
    end if;
    swap( A[i], A[j]);
  end while;
  swap( A[i], A[r]);
  return i;
}
```

```
func int quantile(A array;
                  k, l, r int) {
  if r≤l then
    return A[l];
  end if;
  pos := divide( A, l, r);
  if (k ≤ pos-l) then
    return quantile(A, k, l, pos-1);
  else
    return quantile(A, k-pos+l, pos, r);
  end if;
}
```

Analysis

```
func int quantile(A array;
                  k, l, r int) {
  if r≤l then
    return A[l];
  end if;
  pos := divide( A, l, r);
  if (k < pos-l) then
    return quantile(A, k, l, pos-1);
  else if (k > pos-l)
    return quantile(A, k-pos+l, pos, r);
  else
    return A[k];
  end if;
}
```

- Worst-case: Assume arbitrarily badly chosen pivot elements
- pos always is r-1 (or l+1)
- Gives O(n^2)
- Need to chose the pivot element p more carefully

Choosing p

- Assume we can chose p such that we always continue with at most q% of A
  - Reducing by "q%" means: Extend of reduction depends on n
- We perform at most T(n) = T(q*n) +c*n comparisons
  - T(q*n) – recursive descent, with T(0)=0
  - c*n – function "divide"
- $T(n) = T(q*n)+c*n = T(q^2*n)+q*c*n+c*n = T(q^2n)+(q+1)*c*n = T(q^3n)+(q^2+q+1)*c*n = \ldots$

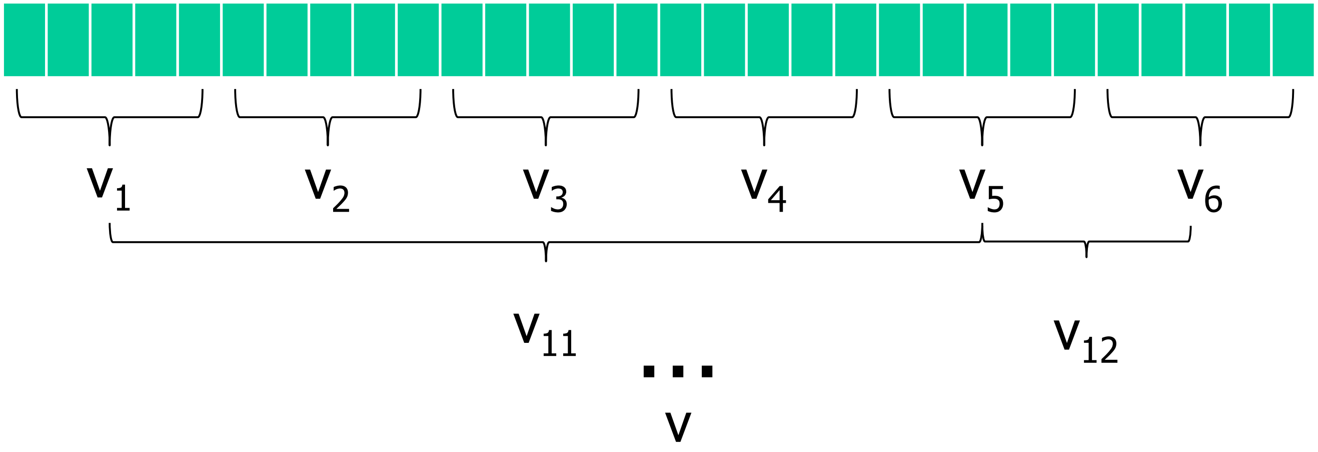
$$\underset{n\to\infty}{T(n)} = c*n*\sum_{i=0}^{n} q^i \le c*n*\sum_{i=0}^{\infty} q^i = c*n*\frac{1}{1-q} = O(n)$$

Discussion

- Our algorithm has worst-case complexity O(n) when we manage to always reduce the array by a fraction of its size, no matter how large the fraction
  - This is not an average-case. We must always (not on average) cut some fraction of A
- Eh – magic?
- No – follows from the way we defined complexity and what we consider as input
- Many operations become “hidden” in the linear factor
  - q=0.9: c*10*n
  - q=0.99: c*100*n
  - q=0.999: c*1000*n

Median-of-Median

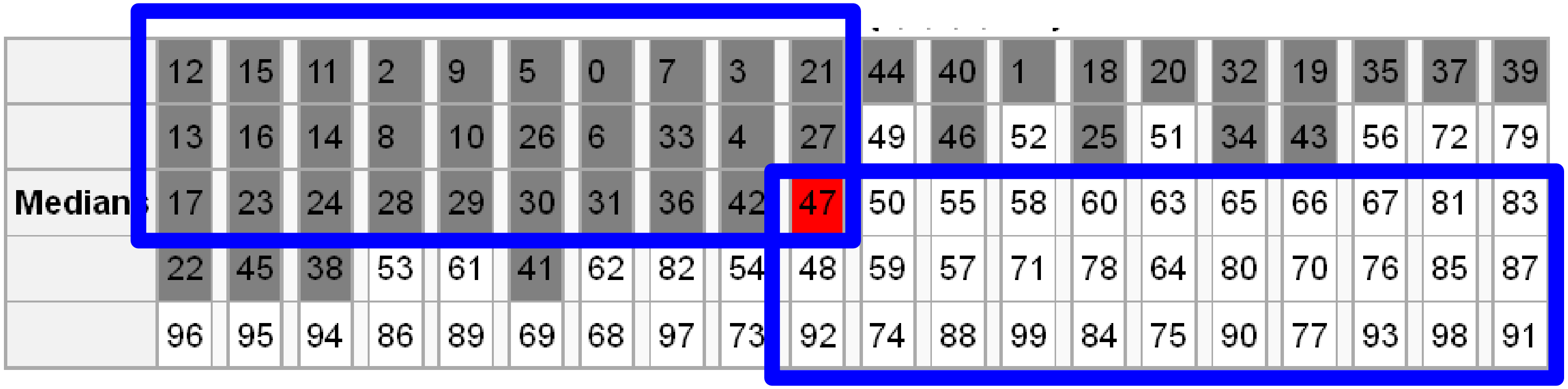
- How can we guarantee to always cut a fraction of A?
- Median-of-median algorithm
  - Partition A in stretches of length 5
  - Compute the median v_i for each partition (with i<floor(n/5))
  - Find the median v of all v_i by repeating this process
    - In general, v will not be the exact median of A – but not far away
  - Use v as pivot element for the quantile computation



Complexity

- O(n): Run through A in partitions of length 5
- O(1): Find each median
  - Runtime of sorting a list of length 5 does not depend on n
- The next iteration will work on only 20% of the input
- Since we always reduce the number of values to look at by 80%, this requires O(n) time in total
  - See previous result

Illustration (source: Wikipedia)



	12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	39
	13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	79
Medians	17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
	22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
	96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91

- Median-of-median of a randomly permuted list 0..99
- For clarity, each 5-tuple is sorted (top-down) and all 5-tuples are sorted by median (left-right)
- Gray/white: Values with actually smaller/greater than med-of-med 47
- Blue: Range with certainly smaller / larger values

Why Does this Help?

- We have ~n/5 first-level-medians v_i
- v (as median of medians) is smaller than halve of them and greater than the other halve (both ~n/10 values)
- Each v_i itself is smaller than (and greater than) 2 values
- Since for the smaller (greater) medians this median itself is also smaller (greater) than v, v is larger (smaller) than at least 3*n/10 elements

Main message

- If you break an array always in the middle, you can do this at most O(log(n)) times
- If you break an array always at 1/3 – 2/3, you also can do this at most O(log(n)) times
- What if we partition an array at any fixed fraction of its size and do linear work in each partition, the overall runtime is still linear
  - And not O(n*log(n))
- But: These “tricks” let the linear factors grow very large
  - So large, that the algorithms are slower in practice even for extremely large inputs
  - “Asymptotically faster” becomes “only theoretically faster”