Lecture 3

Divide and Conquer

吴蔚琪 2022.10.28

991《数据结构与算法》考纲

9、算法基础

(1)字符串模式匹配算法。

(2)贪心法、分治法、动态规划的基本概念。

(3)计算复杂度类别的基本概念，NP-Complete 问题。

Divide and Conquer

- Divide a size n problem into a ($a \geq 1$) sub-problems with size n/a.
- Template of Divide & Conquer
  - Step 1: **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - Step 2: **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - Step 3: **Combine** the solutions to the subproblems into the solution for the original problem.

Divide and Conquer

- When to use Divide and Conquer? If you have found:
  - The problem can be easily solved if the scale of the problem is reduced to a certain extent.
  - The problem can be decomposed into several same problems of a smaller scale, that is, the problem has the optimal substructure property.
  - The solutions of the sub-problems decomposed by this problem can be merged into the solutions of this problem.
  - The sub-problems decomposed by this problem are independent of each other, that is, there are no common sub-problems among them.

Divide and Conquer

- The key point:
  - The keyword: “Divide and Conquer”, “divide”, “merge”, “recursive”, ...
  - The recursive function: 4 methods to solve it.
    - (1) Induction method (Guess, Prove)
    - (2) Substitution method
    - (3) Recursion Tree
    - (4) Master Theorem

Divide and Conquer

- Examples you may have learned during CS101/CS240...
  - Merge Sort
  - Quick Sort
  - Counting Inversions
  - Integer Multiplication
  - Matrix Multiplication
  - Median of Medians
  - Closest Pairs of Points

Median and Selection

Selection

Selection. Given n elements from a totally ordered universe, find k^{th} smallest.

- Minimum: $k = 1$; maximum: $k = n$.
- Median: $k = \lfloor (n + 1) / 2 \rfloor$.
- $O(n)$ compares for min or max.
- $O(n \log n)$ compares by sorting.
- $O(n \log k)$ compares with a binary heap. ← max heap with k smallest

- Goal: solve Select and hence median in O(n) time

Selection

- Algorithm Overview
  - Break the n elements array A into many small groups
  - Find the median of each small group directly
  - Recursively find the median of this set of medians (using Select)
  - Use the median-of-medians to partition A
  - Keep looking for target in one of the two partitions

Select Algorithm 1: QuickSelect

- Pick an arbitrary pivot.
- Partition the array into three pieces.
  - the elements less than the pivot - L
  - the elements equal to the pivot - M
  - the elements that are greater than the pivot - R
- Recursively called on the piece of the array that still contains the kth smallest element.

Select Algorithm 1: QuickSelect

Average-case Running Time

QUICK-SELECT(A, k)

Pick pivot $p \in A$ uniformly at random.

$(L, M, R) \leftarrow$ PARTITION-3-WAY(A, p). ← $\Theta(n)$

IF $(k \leq |L|)$ RETURN QUICK-SELECT(L, k). ← $T(i)$

ELSE IF $(k > |L| + |M|)$ RETURN QUICK-SELECT$(R, k - |L| - |M|)$ ← $T(n - i - 1)$

ELSE IF $(k = |L|)$ RETURN p.

Select Algorithm 1: QuickSelect

Average-case Running Time

Intuition. Split candy bar uniformly ⇒ expected size of larger piece is ¾.

$$T(n) \le T(3n/4) + n \quad \Rightarrow \quad T(n) \le 4n$$

not rigorous: can't assume $\mathbf{E}[T(i)] \le T(\mathbf{E}[i])$

Def. $T(n, k)$ = expected # compares to select k^{th} smallest in array of length $\le n$.

Def. $T(n) = \max_k T(n, k)$.

Proposition. $T(n) \le 4n$.

Pf. [by strong induction on n]

- Assume true for $1, 2, \ldots, n-1$.
- $T(n)$ satisfies the following recurrence:

can assume we always recur of larger of two subarrays since $T(n)$ is monotone non-decreasing

$$\begin{aligned} T(n) &\le n + 1/n\,[\; 2T(n/2) + \ldots + 2T(n-3) + 2T(n-2) + 2T(n-1)\;] \\ &\le n + 1/n\,[\; 8(n/2) + \ldots + 8(n-3) + 8(n-2) + 8(n-1)\;] \\ &\le n + 1/n\,(3n^2) \\ &= 4n. \quad \blacksquare \end{aligned}$$

inductive hypothesis

tiny cheat: sum should start at $T(\lfloor n/2 \rfloor)$

Select Algorithm 1: QuickSelect

Worst-case Running Time

- Question: Consider the array A = [1, 2, . . . , n] shuffled into some arbitrary order. What is the worst-case runtime of Quickselect(A, n/2) in terms of n? Construct an sequence of pivot which have the worst run-time.

A single partition takes $O(n)$ time on an array of size n. The worst case would be if the partition times were $n + (n-1) + \cdots + 2 + 1 = O(n^2)$.
This would happen if the pivot choices were $1, 2, 3, \cdots, \lfloor n/2 \rfloor - 1, n, (n-1), \cdots, \lfloor n/2 \rfloor$. In each of these cases, the partition happens so that one piece only has one element, and the other piece has all the other elements. To get a better runtime, we would like the pieces to be move balanced.

- What if we want to ensure the worst-case runtime is also in O(n)?

Select Algorithm 2: Median of Medians

- Divide n elements into ⌊n / 5⌋ groups of 5 elements each (plus extra).
- Find median of each group (except extra).
- Find median of ⌊n / 5⌋ medians recursively.
- Use median-of-medians as pivot element.

MOM-SELECT(A, k)

$n \leftarrow |A|$.

IF ($n < 50$)

  RETURN k^{th} smallest of element of A via mergesort.

Group A into $\lfloor n / 5 \rfloor$ groups of 5 elements each (ignore leftovers).

$B \leftarrow$ median of each group of 5.

$p \leftarrow$ MOM-SELECT(B, $\lfloor n / 10 \rfloor$) ⟵ median of medians

(L, M, R) ← PARTITION-3-WAY(A, p).

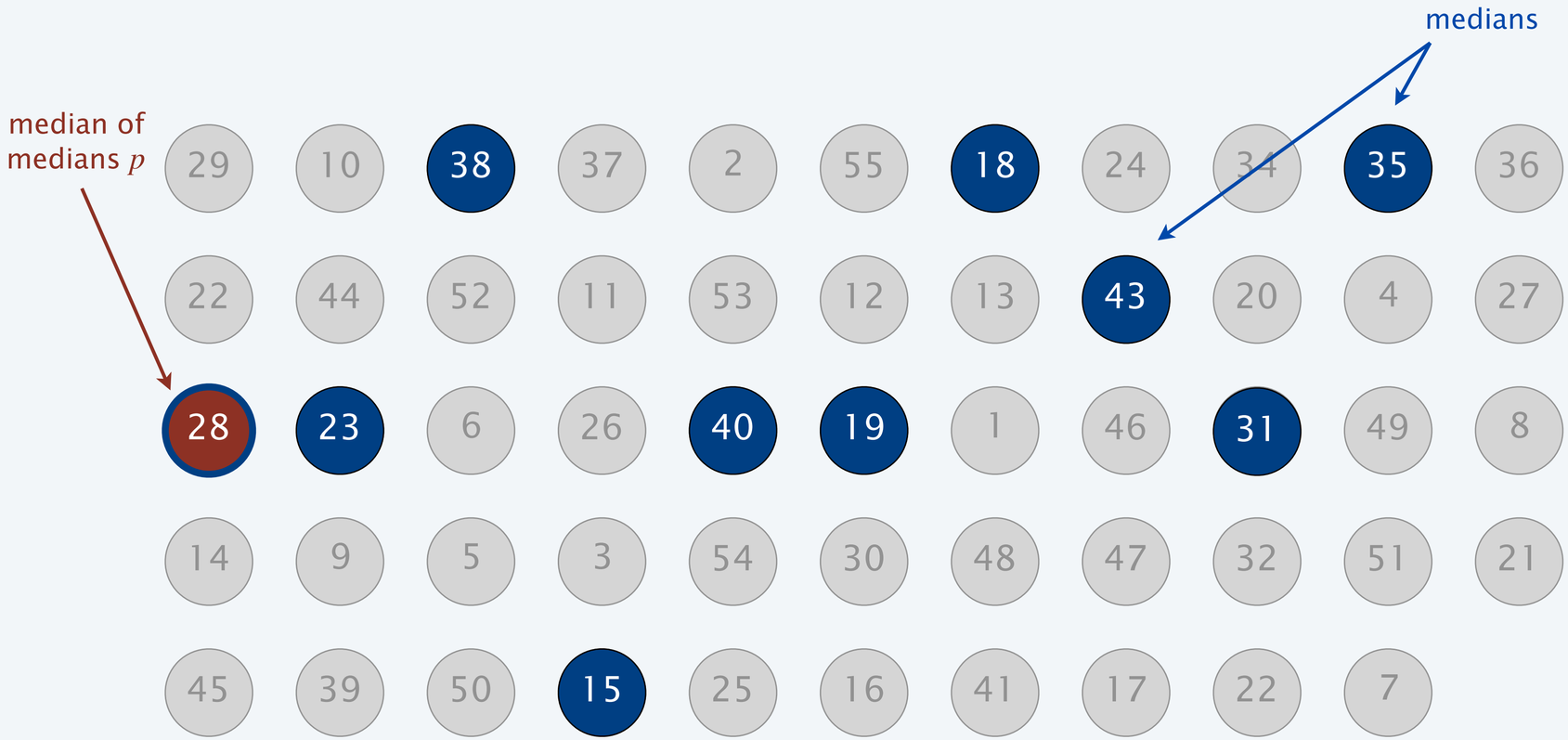
IF ($k \le |L|$) RETURN MOM-SELECT(L, k).

ELSE IF ($k > |L| + |M|$) RETURN MOM-SELECT(R, $k - |L| - |M|$)

ELSE RETURN p.

Analysis of median-of-medians selection algorithm

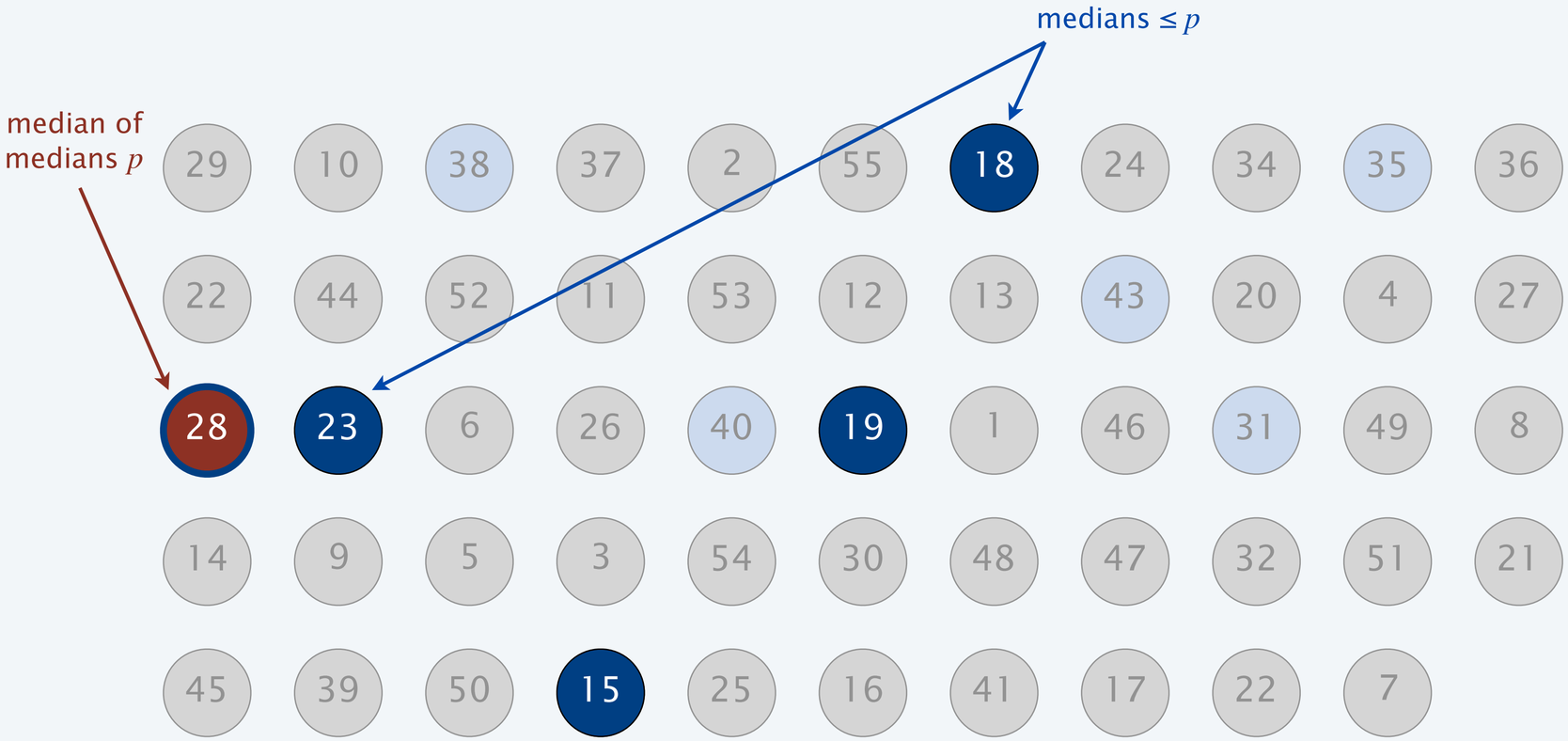
- At least half of 5-element medians ≤ p.



n = 54

Analysis of median-of-medians selection algorithm

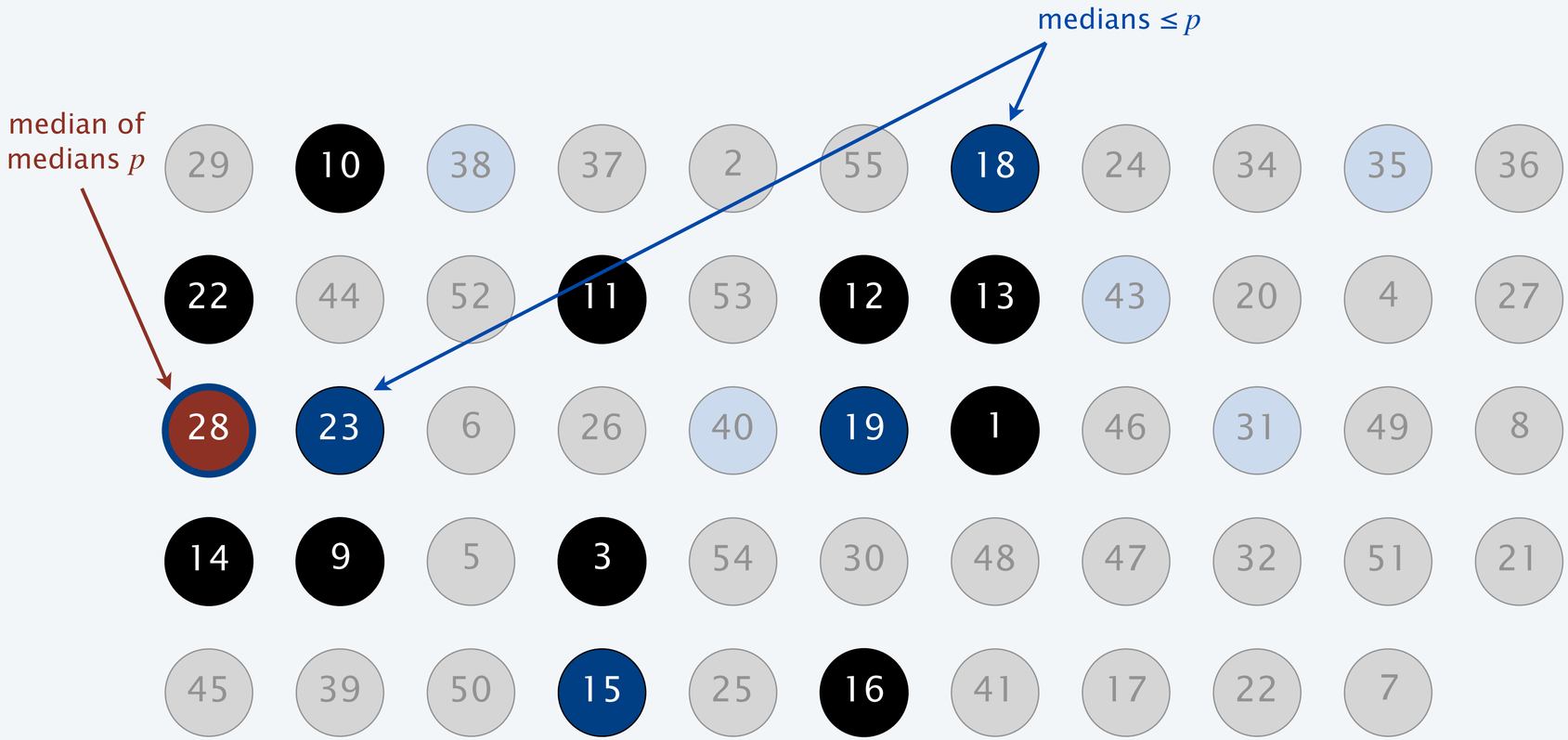
- At least half of 5-element medians $\leq p$.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians $\leq p$.



n = 54

Analysis of median-of-medians selection algorithm

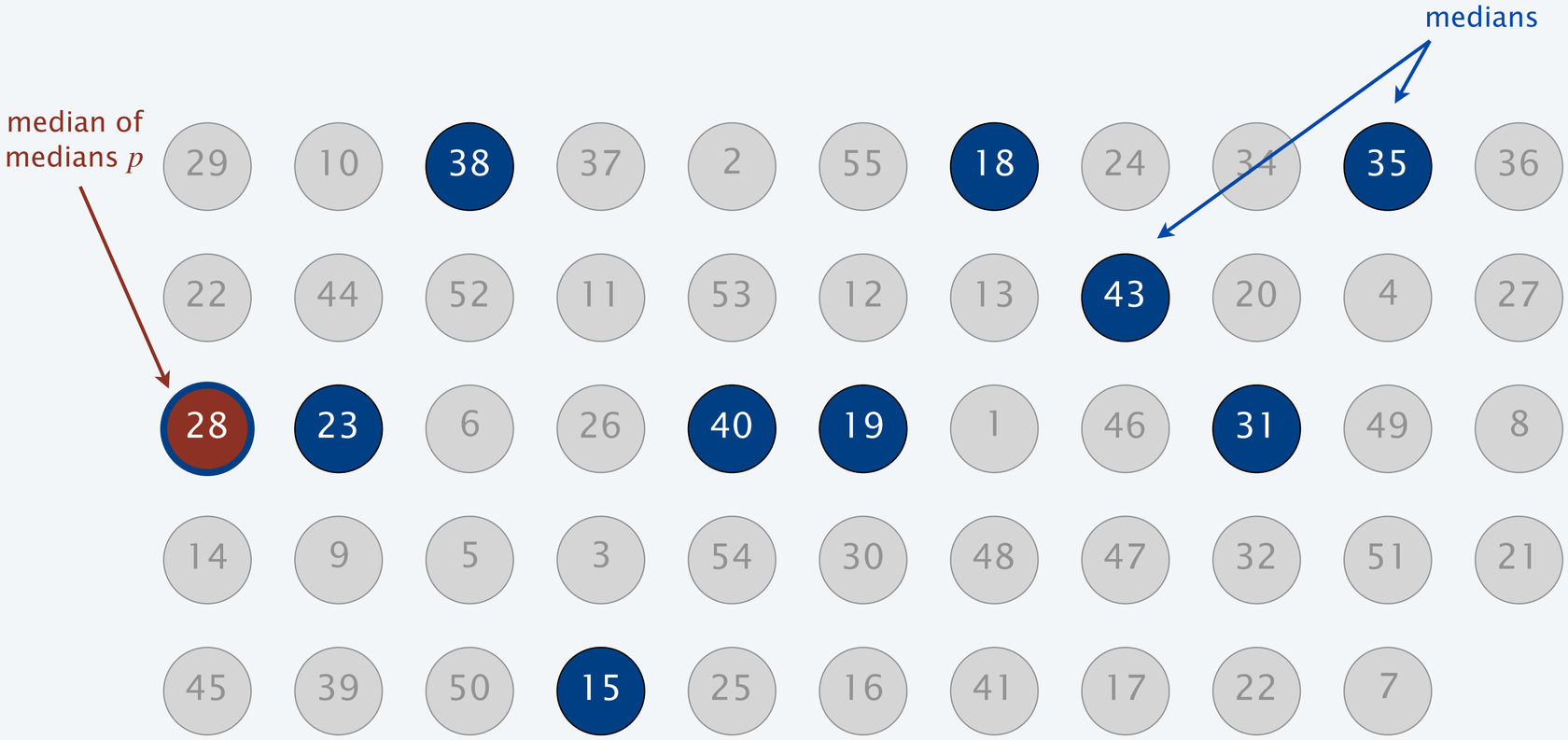
- At least half of 5-element medians $\leq p$.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians $\leq p$.
- At least $3 \lfloor n / 10 \rfloor$ elements $\leq p$.



n = 54

Analysis of median-of-medians selection algorithm

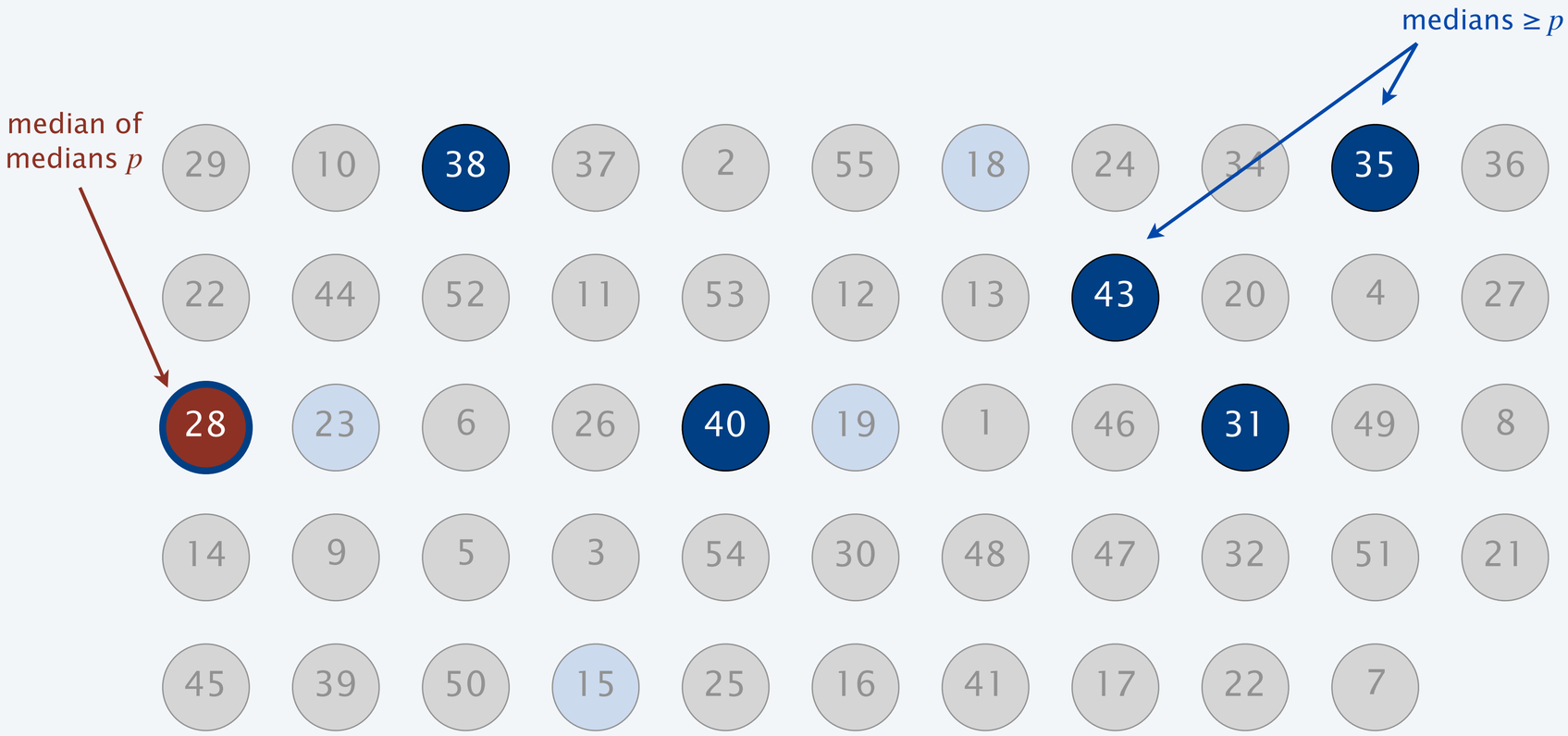
- At least half of 5-element medians ≥ p.



n = 54

Analysis of median-of-medians selection algorithm

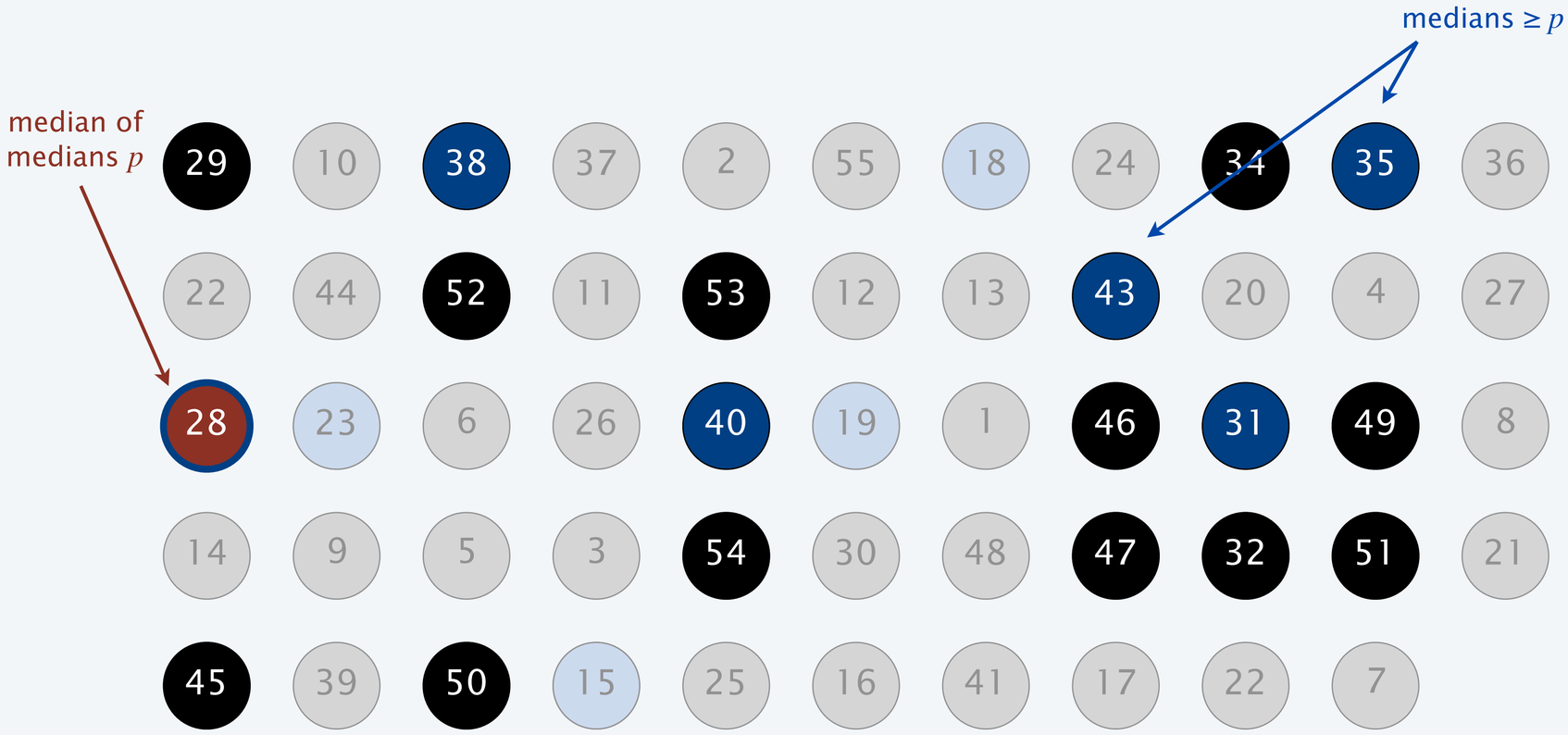
- At least half of 5-element medians ≥ p.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians ≥ p.



n = 54

Analysis of median-of-medians selection algorithm

- At least half of 5-element medians ≥ p.
- At least $\lfloor \lfloor n / 5 \rfloor / 2 \rfloor = \lfloor n / 10 \rfloor$ medians ≥ p.
- At least $3 \lfloor n / 10 \rfloor$ elements ≥ p.



n = 54

Select Algorithm 2: Median of Medians

- And then we prove that for the chosen pivot...

Let p be the chosen pivot. Show that for least $3n/10$ elements x we have that $p \geq x$, and that for at least $3n/10$ elements we have that $p \leq x$.

Let the choice of pivot be p. At least half of the groups $(n/10)$ have a median m such that $m \leq p$. In each of these groups, 3 of the elements are at most the median m (including the median itself). Therefore, at least $3n/10$ elements are at most the size of the median. The same logic follows for showing that $3n/10$ elements are at least the size of the median.

- How does this observation helps with the worst-case running time?

Select Algorithm 2: Median of Medians

- Show that the worst-case runtime of Median-of-mediansSelection(A, k) is O(n).

We end up with the following recurrence:

$$T(n) \leq \underbrace{T(n/5)}_{(A)} + \underbrace{T(7n/10)}_{(B)} + \underbrace{d \cdot n}_{(C)}$$

(A) Calling Better-Quickselect to find the median of the array of medians

(B) The recursive call to Better-Quickselect after performing the partition. The size of the partition piece is always at most $7n/10$ due to the property proved in the previous part.

(C) The time to construct the array of medians, and to partition the array after finding the pivot. This is $O(n)$, but we explicitly write that it is $d \cdots n$ for convenience in the next part.

Select Algorithm 2: Median of Medians

- Show that the worst-case runtime of Median-of-mediansSelection(A, k) is O(n).
- By induction:

For the inductive case,

$$\begin{aligned} T(n) &\leq T(n/5) + T(7n/10) + d \cdot n \\ &\leq c(n/5) + c(7n/10) + d \cdot n \\ &\leq (\frac{9}{10}c + d)n \end{aligned}$$

We pick c large enough so that $(\frac{9}{10}c + d) \leq c$, i.e $c \geq 10d$. Then we can get $T(n) \leq cn$ for constant c, $T(n) = O(n)$

Merged Median

Merged Median

- Given *k* sorted arrays of length *l*, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the $n = kl$ elements. Your algorithm should run asymptotically faster than $O(n)$.
- Examples:
  - a = [1, 3, 5, 7], b = [1, 2, 3], c = [2, 4]
  - merged: 1, 1, 2, 2, 3, 3, 4, 5, 7
  - merged median: 3

Algorithm 6 $sth(s, a_1[1, \cdots, n_1], a_2[1, \cdots, n_2], \cdots, a_k[1, \cdots, n_k])$

if $n_1 \leq 1$ and $n_2 \leq 1$ and $\cdots$ and $n_k \leq 1$ **then**
  $a \leftarrow Sorted([a_1[1], a_2[1], \cdots, a_k[1]])$
  return $a[s]$
end if
for $i = 1$ to k **do**
  if $n_i \geq 1$ **then**
    $b_i = Median(a_i)$
  end if
end for
$median \leftarrow Weighted\_Median([b_1, b_2, \cdots, b_k, n_1, n_2, \cdots n_k])$ //Weighted median is the function that sort the median first and add up the size of the previous lists until they add up to half.
for $i = 1$ to k **do**
  $c_i = BinarySearch(median, a_i)$
end for
if $Sum(c_i) < s$ **then**
  return $sth(s - Sum(c_i), a_1[c_1, \cdots, n_1], a_2[c_2, \cdots, n_2], \cdots, a_k[c_k, \cdots, n_k])$
else
  return $sth(s, a_1[1, \cdots, c_1], a_2[1, \cdots, c_2], \cdots, a_k[1, \cdots, c_k])$
end if

Proof of Correctness:

Firstly, I assume the end stituation is all the array has only one or no element. Then I can sort these elements and get the sth from these.

Secondly, I got the weighted medians from the k arrays. And using binary search to get how many elements in every array in smaller than median of medians. If the total number of the elements which is smaller than median is smaller than s, then we just erase them and find the $(s - sum)$th element in the rest arrays, since the sth element is sure not smaller than median, so we just need to look it up in the rest of the arrays. Else, we just erase the element larger than s, then find the sth element in the rest arrays, since the sth element is smaller than median, so we just need to look it up in the rest of the arrays.

Runtime Analysis:

Since the weighted median is between the $[n/4, 3n/4]$ items, and the weighted median algorithm has $O(k)$ complexity, the binary search algorithm has $O(k \log l)$ complexity.

So $T(n) \leq O(k \log l) + T(3n/4)$. And the recursion can be over in $O(\log l)$.

So $T(n) = O(k \log^2 l)$

More about Solving Recursive Function...

Solving Recursive Function

- Mainly three methods:
  - Substitution method
    - Step 1: Guess the form of the solution.
    - Step 2: Use mathematical induction to find the constants and show that the solution works.
  - Recursive Tree
  - Master Theorem

Solving Recursive Function: Substitution Method

- Example: T(1) = 1, T(n) = 2T(n−1) + c1
- Solution:
  - Expanding the equation out a bit
    T(n) = 2T(n-1) + c1 = 2(2T(n-2) + c1) + c1 = 2(2(2T(n-3) + c1)+ c1) + c1 …
  - Guess that T(n) will be O(2^n)
  - Guess an upper bound k*2^n – b, where k and b are constants (to deal with the c1)

Solving Recursive Function: Substitution Method

- Example: T(1) = 1, T(n) = 2T(n−1) + c1
- Solution: what we want to prove now is $T(n) \leq k*2\hat{}n - b$ for some two constants k and b

Base case: $n = 1$. $T(1) = 1 \leq k2^1 - b = 2k - b$. This is true as long as $k \geq (b + 1)/2$.

Inductive case: We assume our property is true for $n - 1$. We now want to show that it is true for n.

$$
\begin{aligned}
T(n) &= 2T(n-1) + c_1 \\
&\leq 2(k2^{n-1} - b) + c_1 \qquad \text{(by IH)} \\
&= k2^n - 2b + c_1 \\
&\leq k2^n - b
\end{aligned}
$$

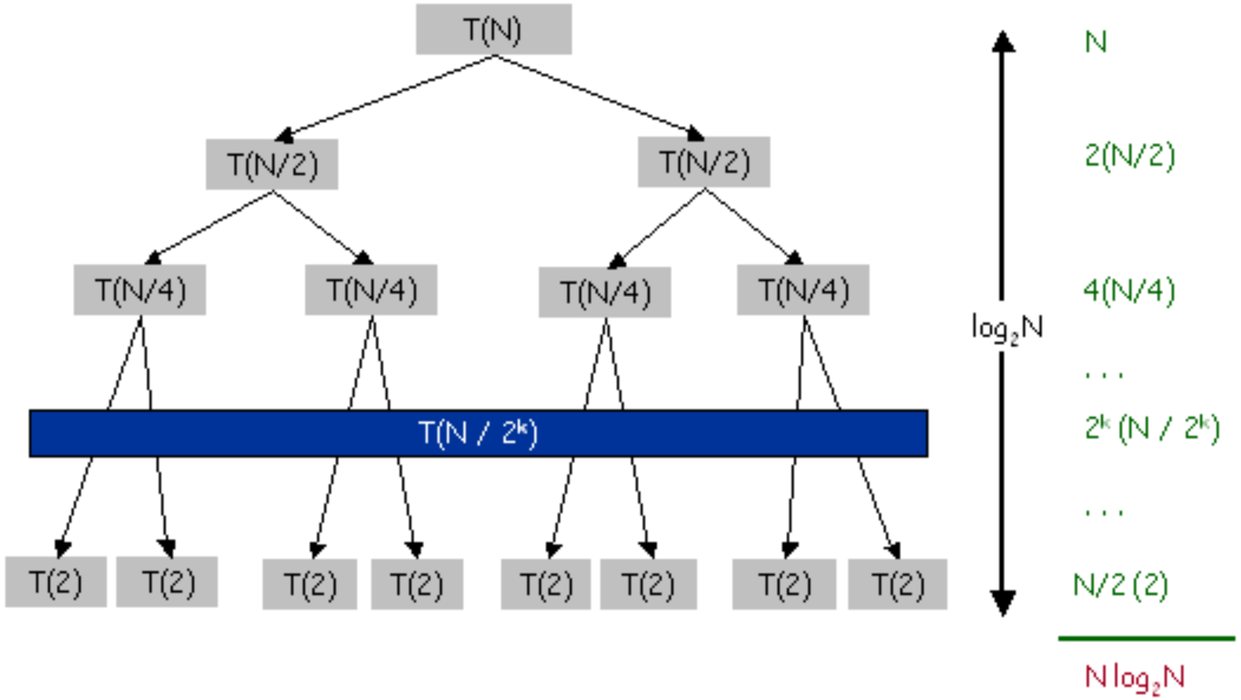
This is true as long as $b \geq c_1$.

Solve Recursive Function: Recursion Tree

- Visualize the recursive calls that occur during mergesort(n).
- There are $\log_2 n$ levels in the recursion tree.
- At level i, there are 2^i recursive calls mergesort($n/2^i$).
- Each call does $n/2^i$ work in merge function.
  - So total work at level i is $2^i * \left(\frac{n}{2^i}\right) = n$.
- So total work overall is $S(n) = n \log_2 n$.

```
void mergesort(n) {
  if (n==1)
    return;
  else {
    L=mergesort(n/2);
    R=mergesort(n/2);
  }
  // takes O(n) time
  merge(R,L);
}
```

- (1) Calculate how many levels.
- (2) Calculate how many recursive calls each level has.
- (3) Calculate how many work does each recursive call does.
  - Then calculate the total work on each level.
- (4) Calculate how many total work you need to do.



Solve Recursive Function: Master Theorem

- Based on comparing the nonrecursive complexity $f(n)$ with $n^{\log_b a}$.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Solve Recursive Function: Master Theorem

- Ex $T(n) = 9T\left(\frac{n}{3}\right) + n$
  - $a = 9, b = 3, f(n) = n, \log_b a = 2.$
  - Check $f(n) = O(n^{2-\epsilon})$, so use case 1 of Master theorem.
  - So $T(n) = \Theta(n^2)$.
- Ex $T(n) = T\left(\frac{2n}{3}\right) + 1.$
  - $a = 1, b = \frac{3}{2}, f(n) = 1, \log_b a = 0.$
  - $f(n) = \Theta(n^0)$, so use case 2 of theorem.
  - So $T(n) = n^0 \log n = \Theta(\log n)$.
- Ex $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$.
  - $a = 3, b = 4, f(n) = n \log n, \log_b a \approx 0.793.$
  - $f(n) = \Omega(n^{0.793+\epsilon})$, so use case 3 of theorem.
  - So $T(n) = \Theta(n \log n)$.

Recurrence relations not solvable by the master method

For example, $T(n) = 2T(n/2) + n/(\log n)$ satisfies all the explicit requirements: we have a=2, b=2, and h(n) = $n/(\log n)$. Nevertheless, it does not fit any of the cases:

- Case 1: $n/(\log n)$ is not $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$, since n^{ε} grows faster than $\log n$
- Case 2: $n/(\log n)$ is not $\Theta(n)$, since $(n)/(n/(\log n)) = \log n$
- Case 3: $n/(\log n)$ is not $\Omega(n^{1+\varepsilon})$ for any $\varepsilon > 0$, since $(n^{1+\varepsilon})/(n/(\log n)) = n^{\varepsilon}\log n$

- Little Trick:
  - Find a similar recurrence that is larger than T(n)
  - Analyze the new recurrence using the master method
  - Use the result as an upper bound for T(n)

Recurrence relations not solvable by the master method

- Little Trick:
  - Find a similar recurrence that is larger than T(n)
  - Analyze the new recurrence using the master method
  - Use the result as an upper bound for T(n)

$T(n) = 2T(n/2) + n/(\log n) \leq 2T(n/2) + n$, so if we call $R(n)$ the function such that $R(n)=2T(n/2)+ n$, we know that $R(n)\geq T(n)$. This is something we can apply the master method to: n is $\Theta(n)$, so $R(n)$ is $\Theta(n \log n)$. Since $T(n) \leq R(n)$, we can conclude that $T(n)$ is $O(n \log n)$. Note that we only end up with O, not Θ; since we were only able apply the master method indirectly, we could not show a tight bound.