# CMPS 2200 – Fall 2017

# *Randomized Algorithms, Quicksort and Randomized Selection*

**Carola Wenk**

Slides by Carola Wenk and Charles Leiserson

# Deterministic Algorithms

Runtime for deterministic algorithms with input size $n$:

- Best-case runtime

  ➔ Attained by one input of size $n$

- Worst-case runtime

  ➔ Attained by one input of size $n$

- Average runtime

  ➔ Averaged **over all possible inputs** of size $n$

# Deterministic Algorithms: Insertion Sort

```
for j=2 to n {
   key = A[j]
   // insert A[j] into sorted sequence A[1..j-1]
   i=j-1
   while(i>0 && A[i]>key){
      A[i+1]=A[i]
      i--
   }
   A[i+1]=key
}
```

- Best case runtime?
- Worst case runtime?

# Deterministic Algorithms: Insertion Sort

Best-case runtime: $O(n)$, input $[1,2,3,\ldots,n]$

➔ Attained by one input of size $n$

- Worst-case runtime: $O(n^2)$, input $[n, n\text{-}1, \ldots,2,1]$

  ➔ Attained by one input of size $n$

- Average runtime : $O(n^2)$

  ➔ Averaged **over all possible inputs** of size $n$

- What kind of inputs are there?
- How many inputs are there?

# Average Runtime

- What kind of inputs are there?
  - Do [1,2,…,$n$] and [5,6,…,$n$+5] cause different behavior of Insertion Sort?
  - No. Therefore it suffices to only consider all permutations of [1,2,…,$n$] .
- How many inputs are there?
  - There are $n$! different permutations of [1,2,…,$n$]

# Average Runtime Insertion Sort: *n*=4

```
for j=2 to n {
    key = A[j]
    // insert A[j] into sorted sequen
    i=j-1
    while(i>0 && A[i]>key){
      A[i+1]=A[i]
      i--
    }
    A[i+1]=key
}
```

- Inputs: 4!=24

| | | | |
|---|---|---|---|
| [1,2,3,4] **0** | [4,1,2,3] **3** | [4,1,3,2] **4** | [4,3,2,1] **6** |
| [2,1,3,4] **1** | [1,4,2,3] **2** | [1,4,3,2] **3** | [3,4,2,1] **5** |
| [1,3,2,4] **1** | [1,2,4,3] **1** | [1,3,4,2] **2** | [3,2,4,1] **4** |
| [3,1,2,4] **2** | [4,2,1,3] **4** | [4,3,1,2] **5** | [4,2,3,1] **5** |
| [3,2,1,4] **3** | [2,1,4,3] **2** | [3,4,1,2] **4** | [2,4,3,1] **4** |
| [2,3,1,4] **2** | [2,4,1,3] **3** | [3,1,4,2] **3** | [2,3,4,1] **3** |

- Runtime is proportional to: 3 + **#times in while loop**
- Best: 3+**0**, Worst: 3+**6**=9, Average: 3+**72**/24 = 6

# Average Runtime: Insertion Sort

- The average runtime averages runtimes over all $n!$ different input permutations
- Disadvantage of considering average runtime:
  - There are still worst-case inputs that will have the worst-case runtime
  - Are all inputs really equally likely? That depends on the application

$\Rightarrow$ **Better:** Use a randomized algorithm

# Randomized Algorithm: Insertion Sort

- **Randomize the order of the input array:**
  - Either prior to calling insertion sort,
  - or during insertion sort (insert random element)
- This makes the runtime depend on a probabilistic experiment (sequence of numbers obtained from random number generator; or random input permutation)

  ⇒Runtime is a random variable (maps sequence of random numbers to runtimes)
- **Expected runtime** = expected value of runtime random variable

# Randomized Algorithm: Insertion Sort

- Runtime is independent of input order ([1,2,3,4] may have good or bad runtime, depending on sequence of random numbers)
- No assumptions need to be made about input distribution
- No one specific input elicits worst-case behavior
- The worst case is determined only by the output of a random-number generator.

$\Rightarrow$ When possible use expected runtimes of randomized algorithms instead of average case analysis of deterministic algorithms

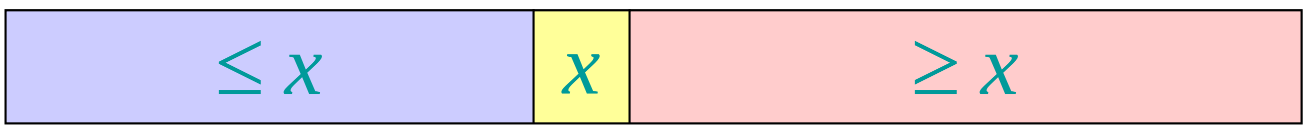
# Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).
- We are going to perform an expected runtime analysis on randomized quicksort

# Quicksort: Divide and conquer

Quicksort an $n$-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** $x$ such that elements in lower subarray $\leq x \leq$ elements in upper subarray.




2. ***Conquer:*** Recursively sort the two subarrays.
3. ***Combine:*** Trivial.

**Key:** *Linear-time partitioning subroutine.*

# Partitioning subroutine

PARTITION$(A, p, q)$ ⊳ $A[p \,..\, q]$

  $x \leftarrow A[p]$ ⊳ pivot $= A[p]$

  $i \leftarrow p$

  **for** $j \leftarrow p + 1$ **to** $q$

    **do if** $A[j] \leq x$

      **then** $i \leftarrow i + 1$

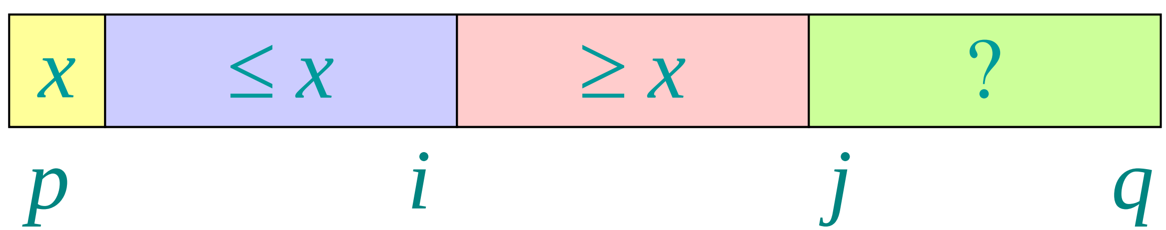
        exchange $A[i] \leftrightarrow A[j]$

  exchange $A[p] \leftrightarrow A[i]$

  **return** $i$

Running time $= O(n)$ for $n$ elements.

***Invariant:***



# Example of partitioning

| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
|---|---|---|---|---|---|---|---|
| $i$ | $j$ | | | | | | |

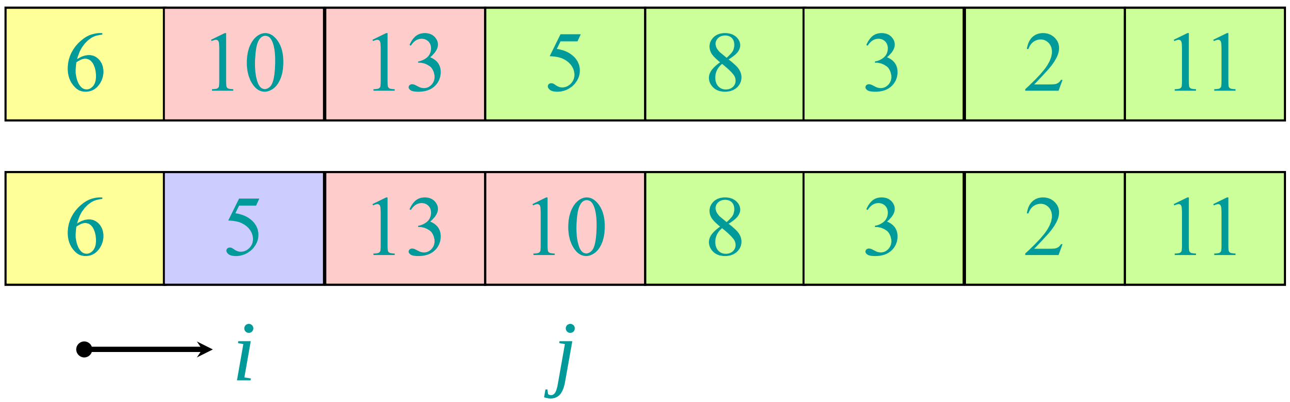
# Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| *i* | | *j* | | | | | |

# Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| $i$ | | | $j$ | | | | |

# Example of partitioning



# Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |

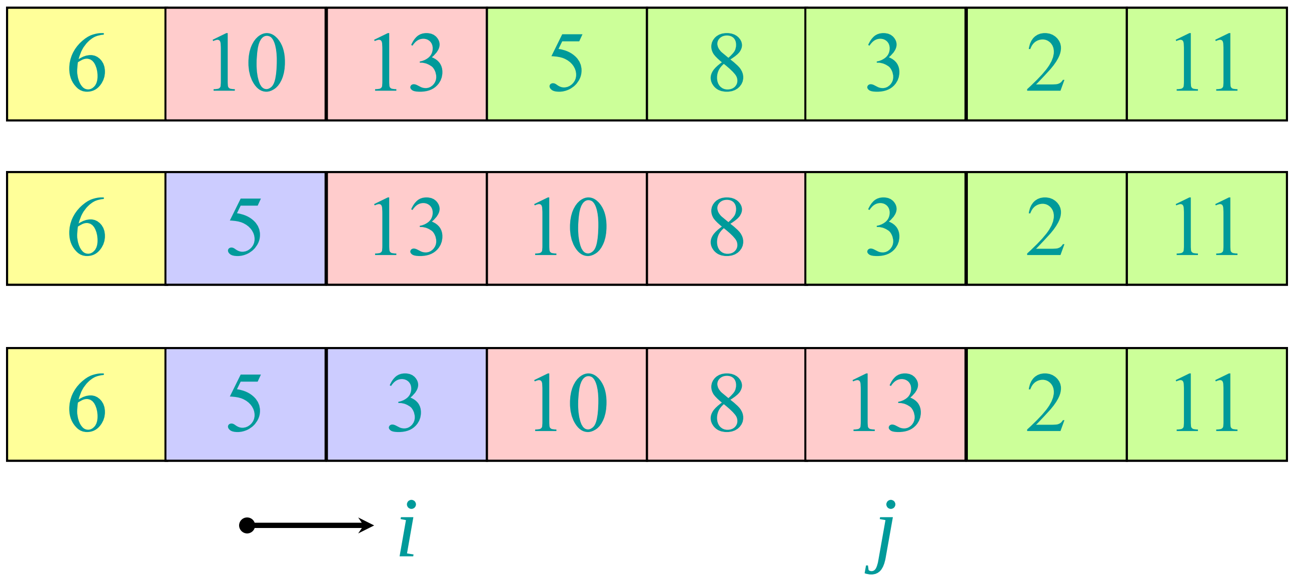
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |

*i* ⟶ *j*

# Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| | *i* | | | ⟶ | *j* | | |

# Example of partitioning



# Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |

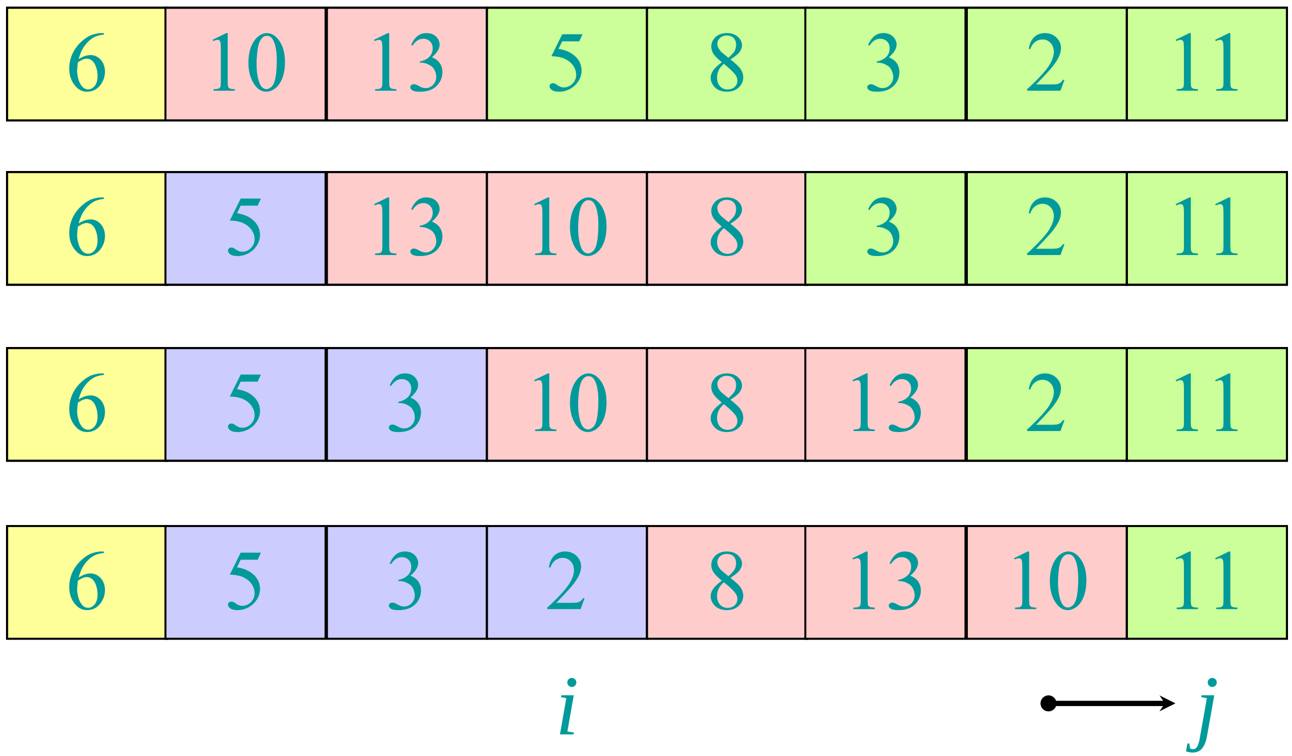
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| | | *i* | | | | → *j* | |

# Example of partitioning

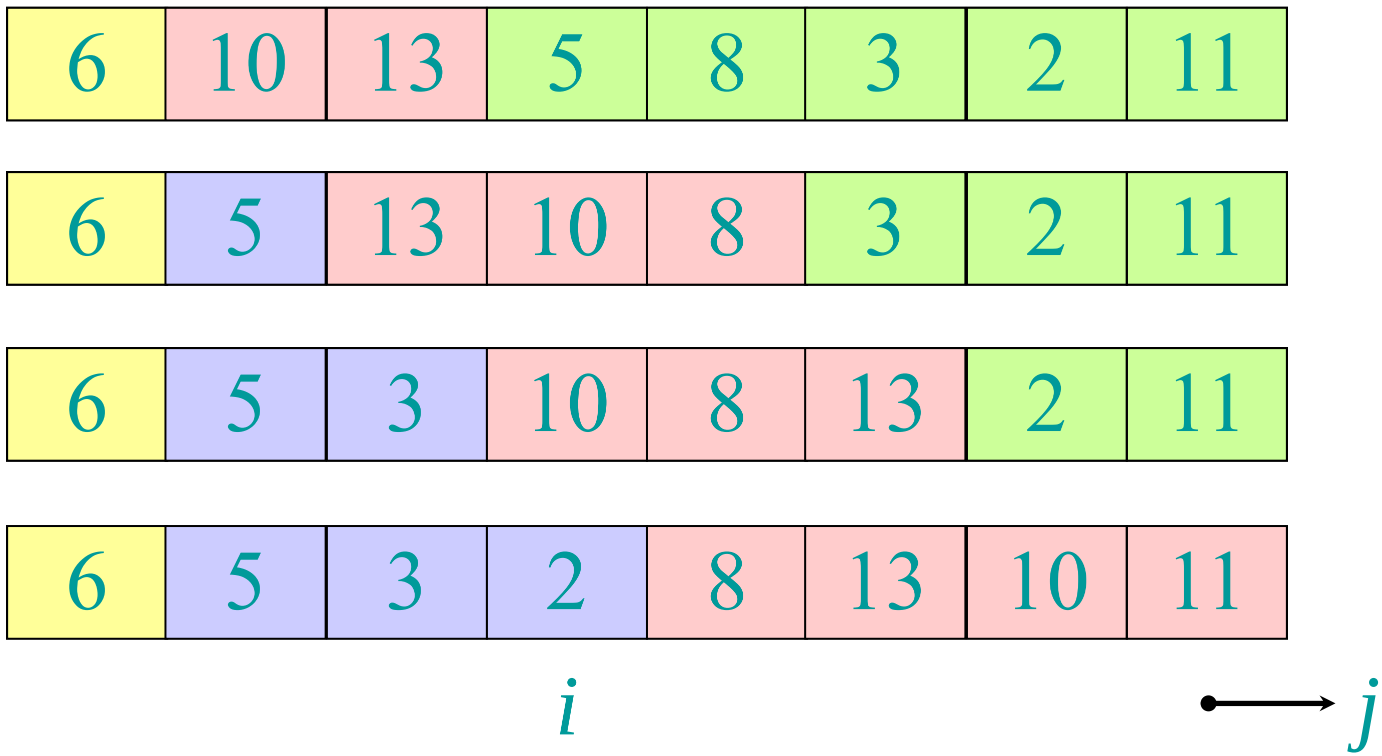
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| 6 | 5 | 3 | 2 | 8 | 13 | 10 | 11 |

→ $i$ $j$

# Example of partitioning



# Example of partitioning



# Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| 6 | 5 | 3 | 2 | 8 | 13 | 10 | 11 |
| 2 | 5 | 3 | 6 | 8 | 13 | 10 | 11 |
| | | | $i$ | | | | |

# Pseudocode for quicksort

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q–1)
         QUICKSORT(A, q+1, r)
```

**Initial call:** QUICKSORT$(A, 1, n)$

# Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of $n$ elements.

# Worst-case of quicksort

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q–1)
         QUICKSORT(A, q+1, r)
```

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

***(arithmetic series)***

# Worst-case recursion tree

$$T(n) = T(0) + T(n–1) + cn$$

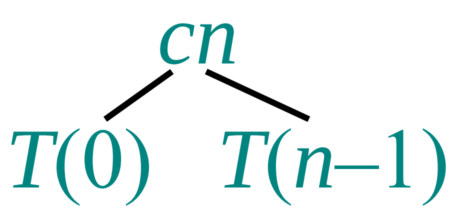
# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

$T(n)$

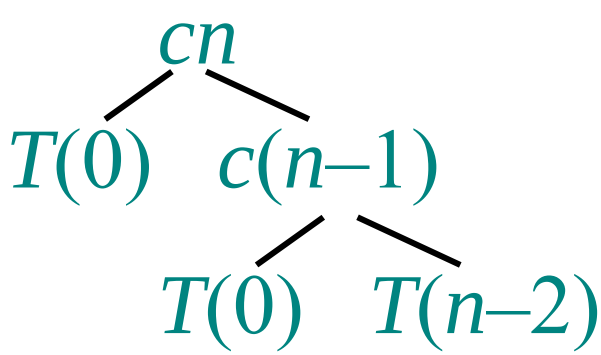
# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



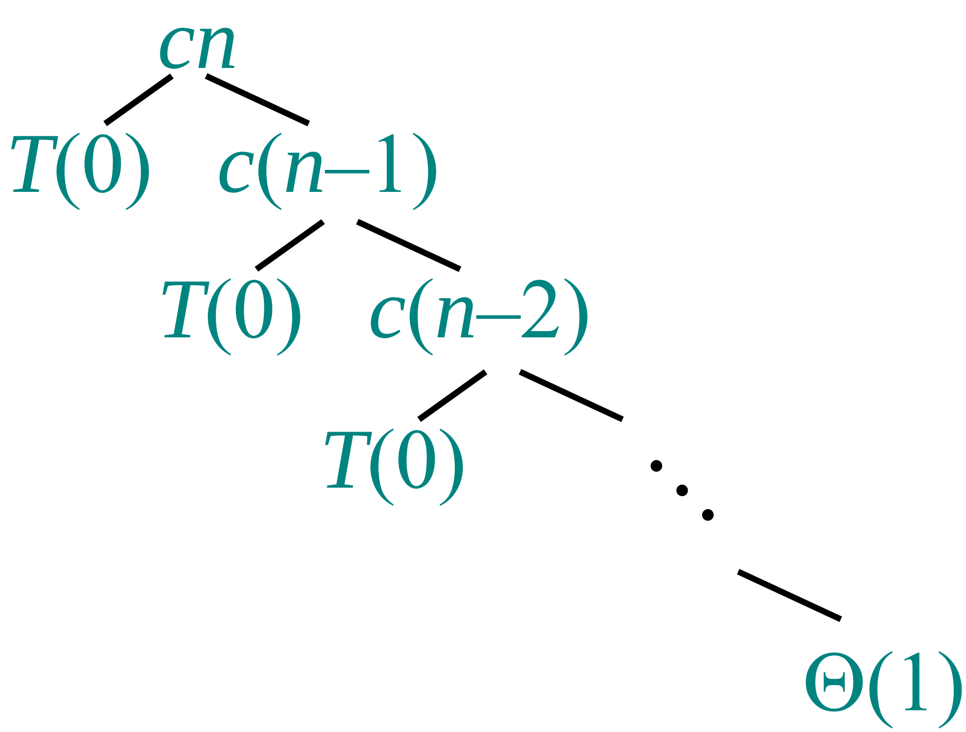
# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



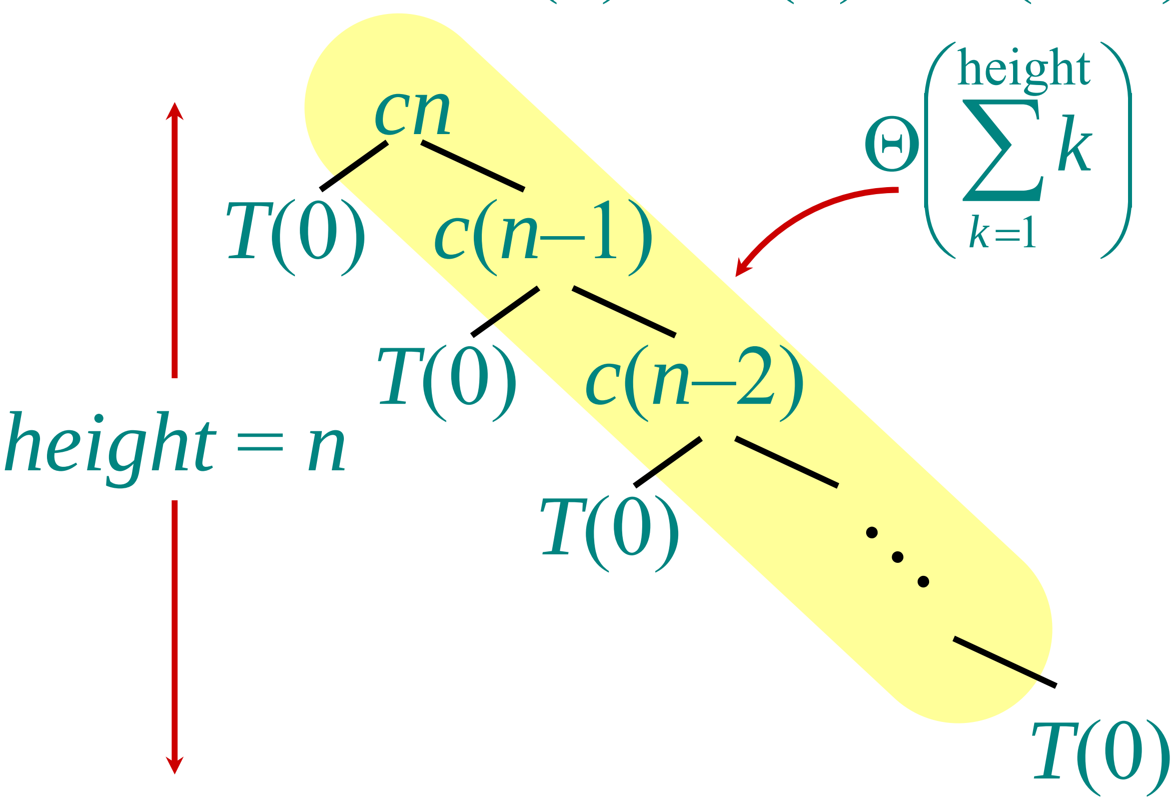
# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



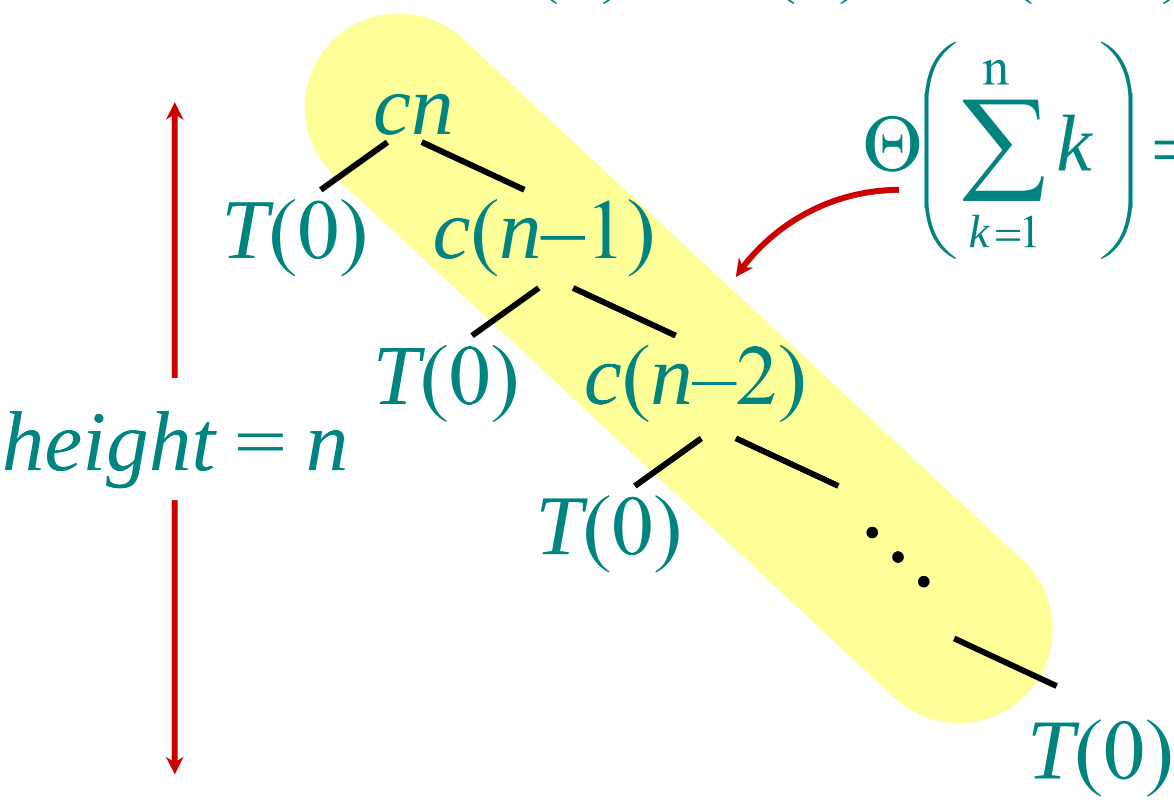
# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



# Worst-case recursion tree

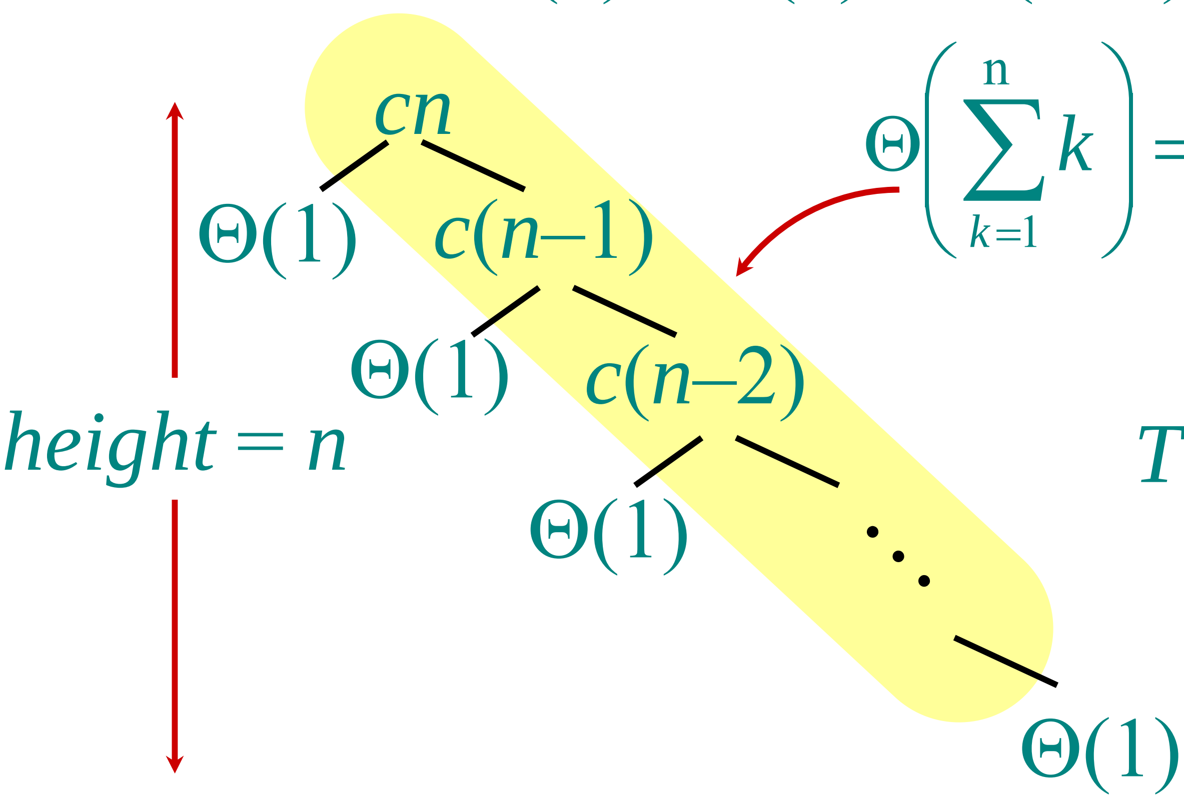
$T(n) = T(0) + T(n–1) + cn$




$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

# Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$




$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$

# Best-case analysis

***(For intuition only!)***

If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \log n) \qquad \text{(same as merge sort)}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

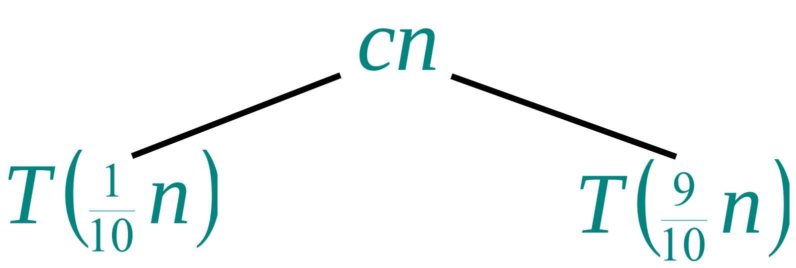
$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

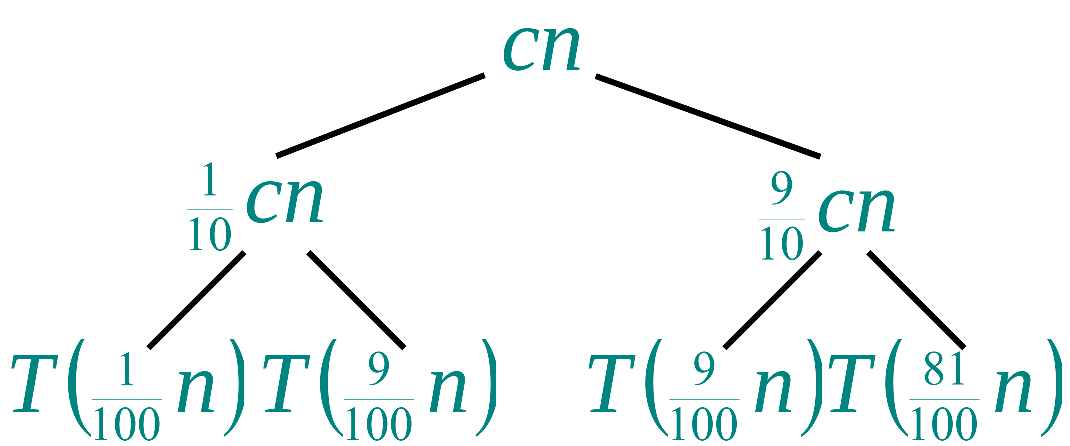
# Analysis of "almost-best" case

$$T(n)$$

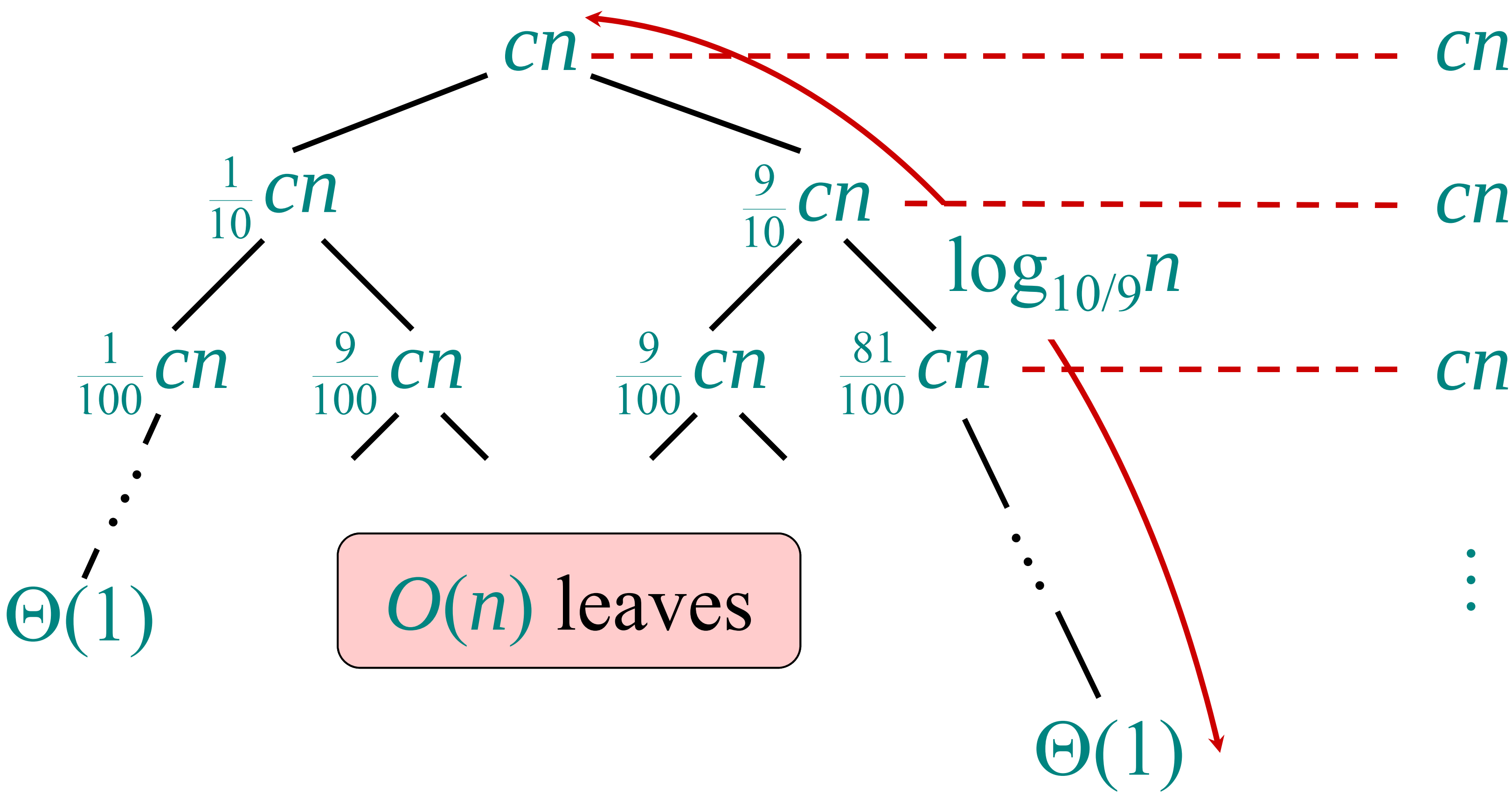
# Analysis of “almost-best” case



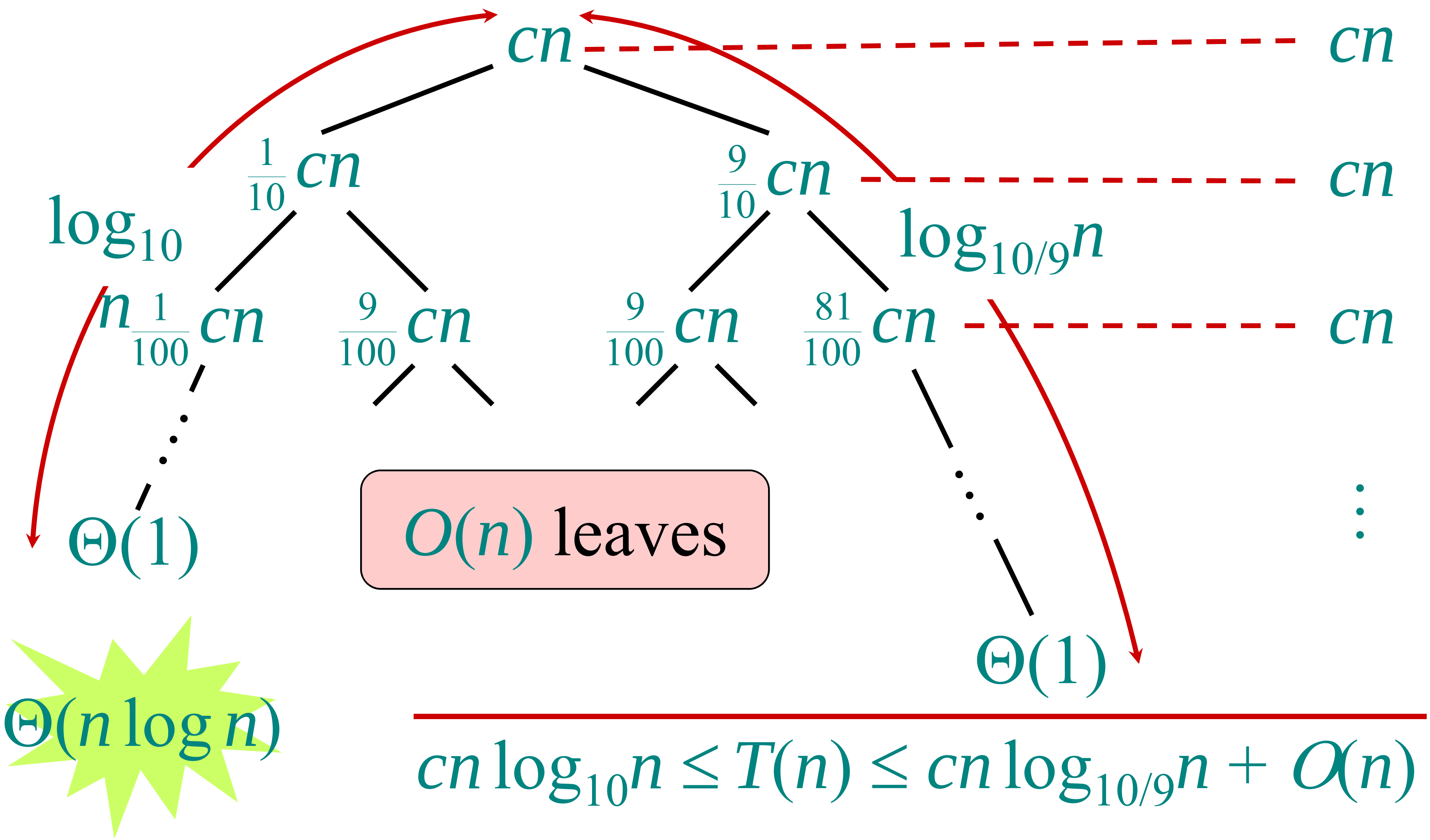
# Analysis of “almost-best” case



# Analysis of "almost-best" case



# Analysis of "almost-best" case




$$cn \log_{10} n \leq T(n) \leq cn \log_{10/9} n + O(n)$$

# Quicksort Runtimes

- Best case runtime $T_{best}(n) \in O(n \log n)$
- Worst case runtime $T_{worst}(n) \in O(n^2)$

- Worse than mergesort? Why is it called quicksort then?
- Its average runtime $T_{avg}(n) \in O(n \log n)$
- Better even, the expected runtime of **randomized quicksort** is $O(n \log n)$

# Average Runtime

The **average runtime** $T_{avg}(n)$ for Quicksort is the average runtime over **all possible inputs** of length n.

- $T_{avg}(n)$ has to average the runtimes over all $n!$ different input permutations.
- There are still worst-case inputs that will have a $O(n^2)$ runtime

⇒ **Better:** Use randomized quicksort

# Randomized quicksort

IDEA: Partition around a ***random*** element.

- Running time is independent of the input order. It depends only on the sequence $s$ of random numbers.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the sequence $s$ of random numbers.

# Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from ***code tuning***.
- Quicksort behaves well even with caching and virtual memory.

# Average Runtime vs. Expected Runtime

- Average runtime is averaged over all inputs of a deterministic algorithm.
- Expected runtime is the expected value of the runtime random variable of a randomized algorithm. It effectively "averages" over all sequences of random numbers.

- De facto both analyses are very similar. However in practice the randomized algorithm ensures that not one single input elicits worst case behavior.

# Order statistics

Select the $i$th smallest of $n$ elements (the element with ***rank*** $i$).

- $i = 1$: ***minimum***;
- $i = n$: ***maximum***;
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: ***median***.

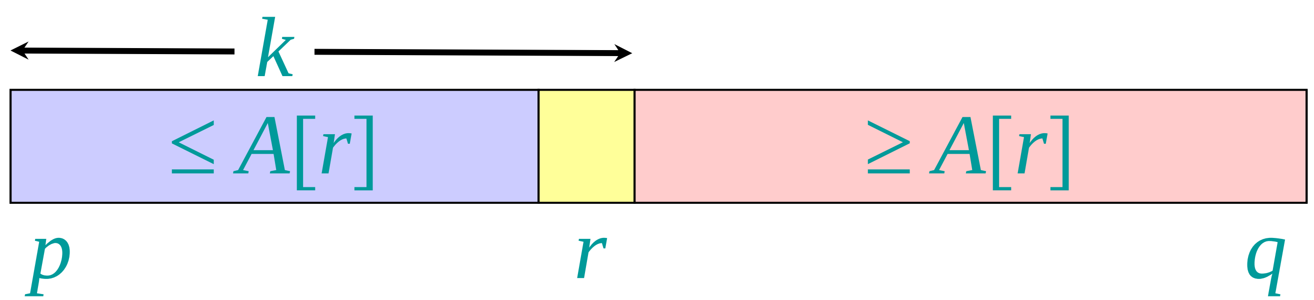
***Naive algorithm***: Sort and index $i$th element.

Worst-case running time $= \Theta(n \log n + 1)$
$= \Theta(n \log n)$,

using merge sort (*not* quicksort).

# Randomized divide-and-conquer algorithm

RAND-SELECT$(A, p, q, i)$ ⊳ $i$-th smallest of $A[p \mathinner{.\,.} q]$
**if** $p = q$ **then return** $A[p]$
$r \leftarrow$ RAND-PARTITION$(A, p, q)$
$k \leftarrow r - p + 1$ ⊳ $k = \text{rank}(A[r])$
**if** $i = k$ **then return** $A[r]$
**if** $i < k$
**then return** RAND-SELECT$(A, p, r - 1, i)$
**else return** RAND-SELECT$(A, r + 1, q, i - k)$



# Example

Select the $i = 7$th smallest:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 | $i = 7$ |

***pivot***

Partition:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 5 | 3 | 6 | 8 | 13 | 10 | 11 | $k = 4$ |

Select the $7 - 4 = 3$rd smallest recursively.

# Intuition for analysis

(All our analyses today assume that all elements are distinct.)

for RAND-PARTITION

**Lucky:**

$T(n) = T(3n/4) + dn$

$= \Theta(n)$

$n^{\log_{4/3} 1} = n^0 = 1$

CASE 3

**Unlucky:**

$T(n) = T(n - 1) + dn$

$= \Theta(n^2)$

arithmetic series

***Worse than sorting!***

# Analysis of expected time

- Call a pivot ***good*** if its rank lies in $[n/4, 3n/4]$.
- How many good pivots are there? $n/2$

  $\Rightarrow$ A random pivot has 50% chance of being good.
- Let $T(n,s)$ be the runtime random variable

time to reduce array size to $\leq 3/4n$

$$T(n,s) \leq T(3n/4,s) + X(s) \cdot dn$$

#times it takes to find a good pivot

Runtime of partition

# Analysis of expected time

**Lemma:** A fair coin needs to be tossed an expected number of 2 times until the first "heads" is seen.

**Proof:** Let $E(X)$ be the expected number of tosses until the first "heads" is seen.

- Need at least one toss, if it's "heads" we are done.
- If it's "tails" we need to repeat (probability ½).

$\Rightarrow E(X) = 1 + \frac{1}{2} E(X)$

$\Rightarrow E(X) = 2$ ∎

# Analysis of expected time

time to reduce array size to $\leq 3/4n$

$$T(n,s) \leq T(3n/4,s) + X(s)\cdot dn$$

#times it takes to find a good pivot

Runtime of partition

$$\Rightarrow E(T(n,s)) \leq E(T(3n/4,s)) + E(X(s)\cdot dn)$$

$$\Rightarrow E(T(n,s)) \leq E(T(3n/4,s)) + E(X(s))\cdot dn$$

*Linearity of expectation*

$$\Rightarrow E(T(n,s)) \leq E(T(3n/4,s)) + 2\cdot dn$$

*Lemma*

$$\Rightarrow T_{exp}(n) \leq T_{exp}(3n/4) + \Theta(n)$$

$$\Rightarrow T_{exp}(n) \in \Theta(n)$$

# Summary of randomized order-statistic selection

- Works fast: linear expected time.
- Excellent algorithm in practice.
- But, the worst case is ***very*** bad: $\Theta(n^2)$.

***Q.*** Is there an algorithm that runs in linear time in the worst case?

***A.*** Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973].

**IDEA:** Generate a good pivot recursively.
This algorithms large constants though and therefore is not efficient in practice.