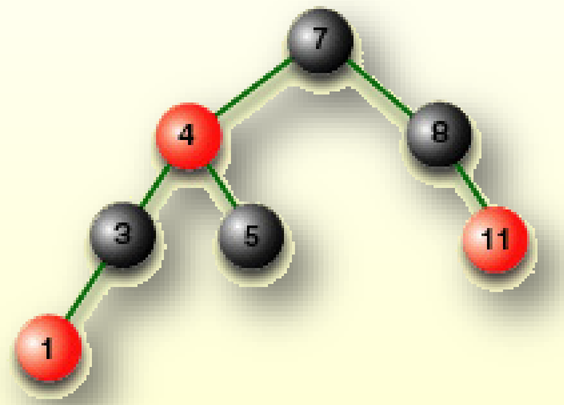
CS161: Design and Analysis of Algorithms



Lecture 4

Leonidas Guibas

Outline

- Review of last lecture
- QuickSort and its analysis
    - Worst-case behavior
    - Best-case behavior
    - Average-case behavior
    - Randomized QuickSort

Slides modified from

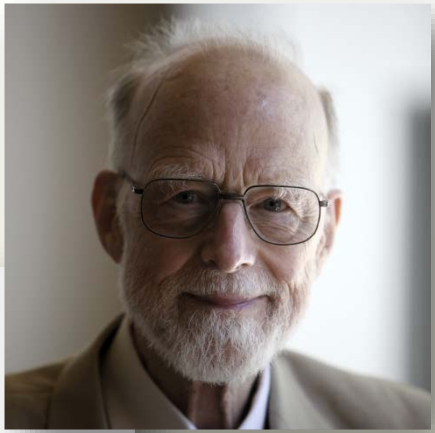
- http://www.cs.virginia.edu/~luebke/cs332/
- http://www.cs.uml.edu/~kdaniels/courses/ALG_404_F10.html
- http://cs.txstate.edu/~rp44/cs3358

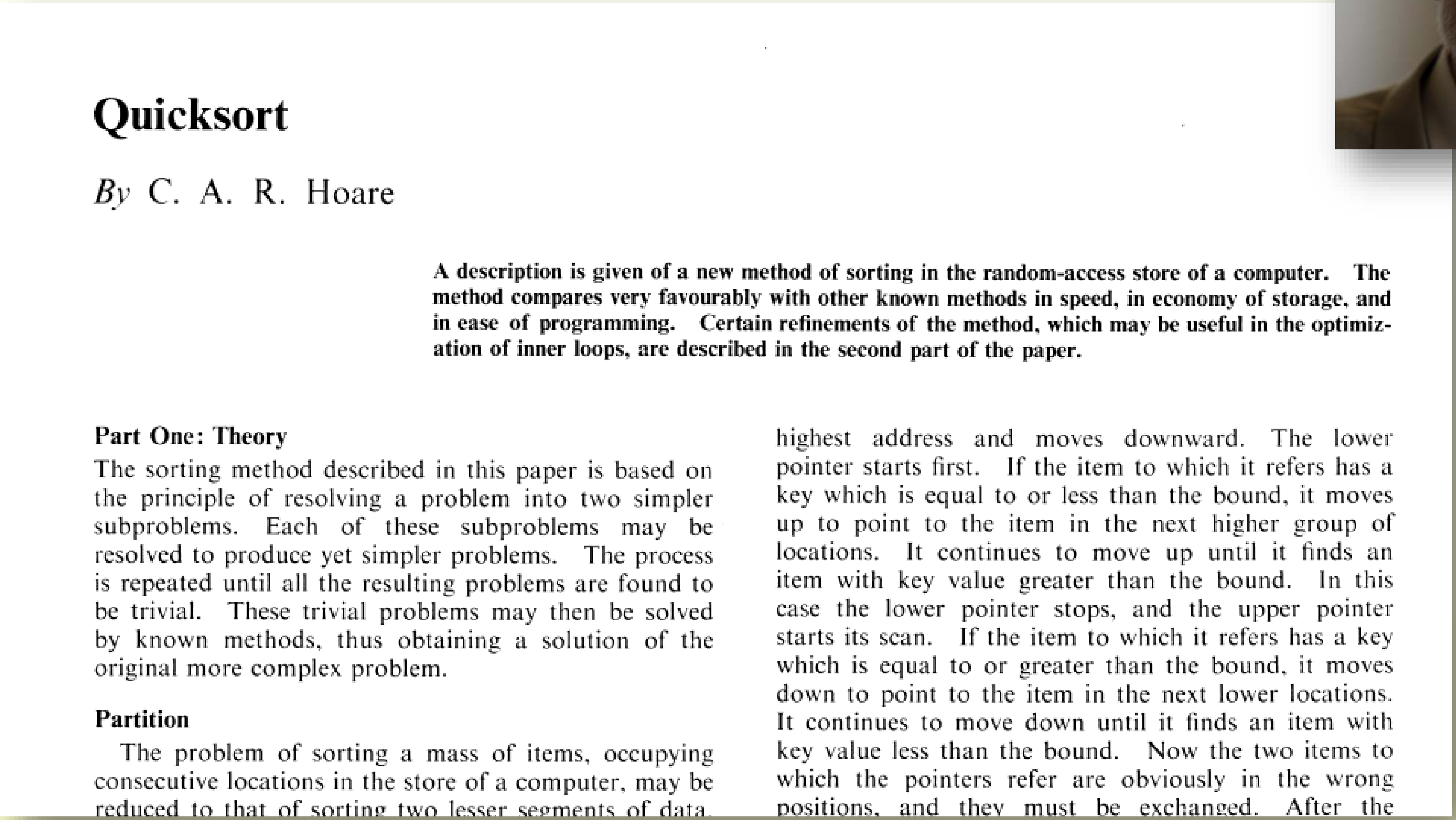
QuickSort

- Another divide and conquer sorting algorithm – like MergeSort
- Excellent performance in practice – often the default sorting package
- Sorts in place – like InsertionSort
- Has many interesting variations

QuickSort

Sir Charles Antony Richard Hoare





Quicksort

By C. A. R. Hoare

A description is given of a new method of sorting in the random-access store of a computer. The method compares very favourably with other known methods in speed, in economy of storage, and in ease of programming. Certain refinements of the method, which may be useful in the optimization of inner loops, are described in the second part of the paper.

Part One: Theory

The sorting method described in this paper is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be solved by known methods, thus obtaining a solution of the original more complex problem.

Partition

The problem of sorting a mass of items, occupying consecutive locations in the store of a computer, may be reduced to that of sorting two lesser segments of data.

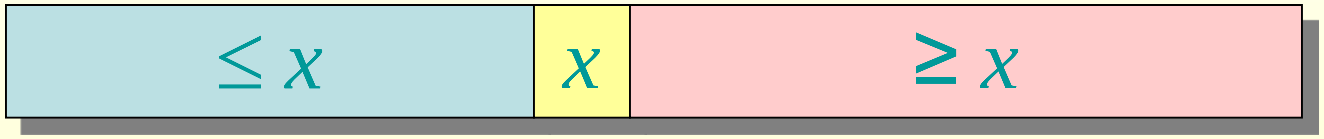
highest address and moves downward. The lower pointer starts first. If the item to which it refers has a key which is equal to or less than the bound, it moves up to point to the item in the next higher group of locations. It continues to move up until it finds an item with key value greater than the bound. In this case the lower pointer stops, and the upper pointer starts its scan. If the item to which it refers has a key which is equal to or greater than the bound, it moves down to point to the item in the next lower locations. It continues to move down until it finds an item with key value less than the bound. Now the two items to which the pointers refer are obviously in the wrong positions, and they must be exchanged. After the

1962

QuickSort

Quicksort on an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x so 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.*

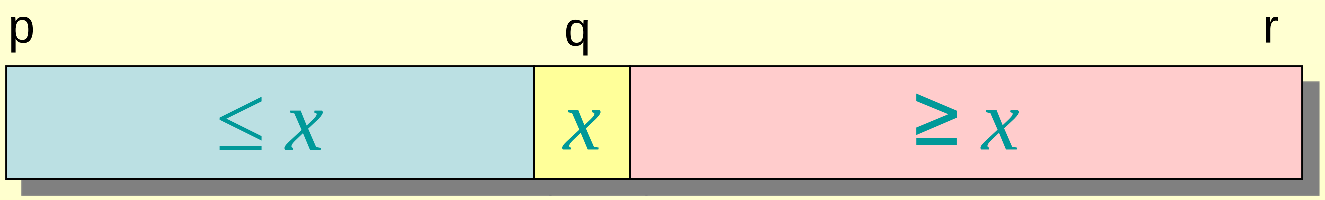
Pseudocode for QuickSort

```
QUICKSORT($A$, $p$, $r$)
    if $p < r$
        then $q \leftarrow$ PARTITION($A$, $p$, $r$)
            QUICKSORT($A$, $p$, $q-1$)
            QUICKSORT($A$, $q+1$, $r$)
```

Initial call: QUICKSORT(A, 1, n)

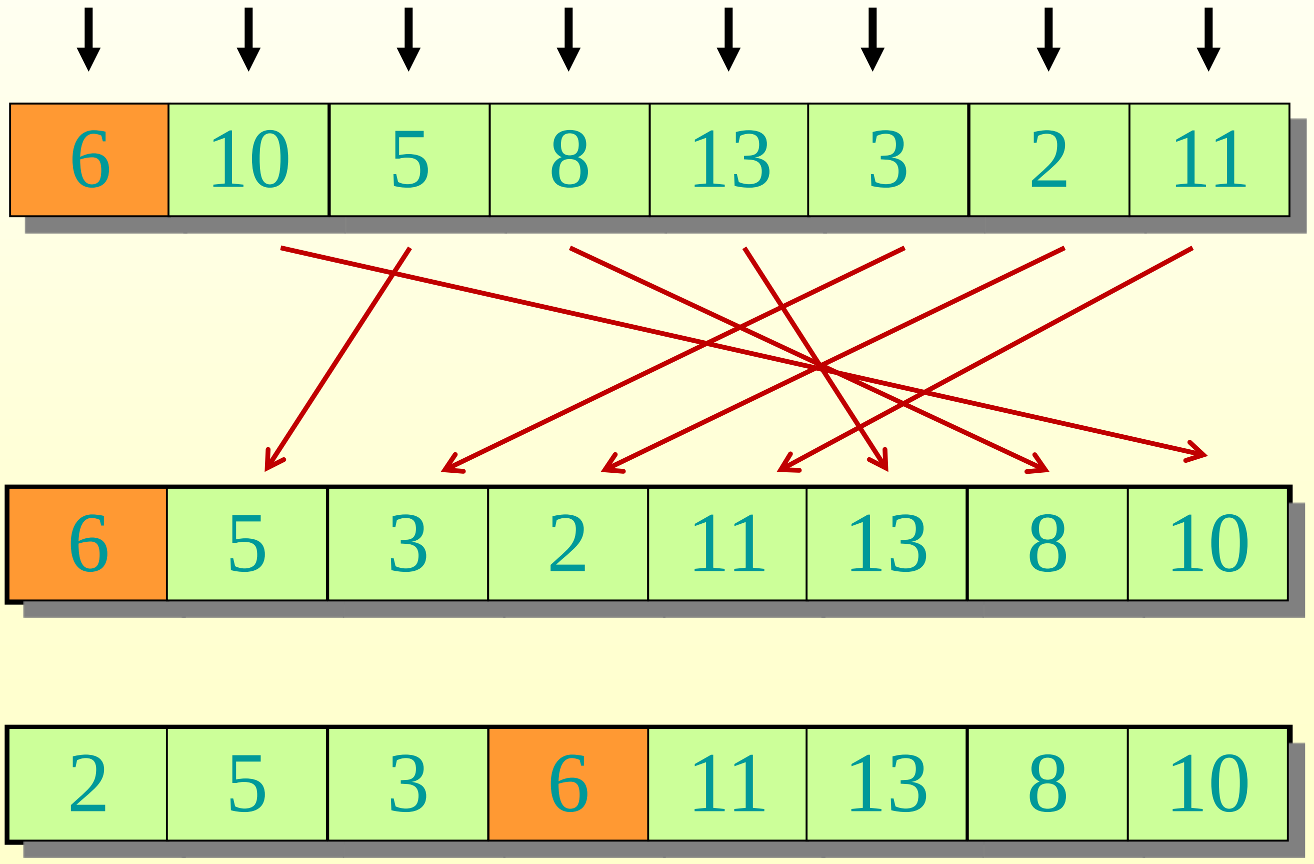
Partition

- All the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result: two subarrays
    - All values in first subarray $\leq$ all values in second
  - Returns the index of the "pivot" element separating the two subarrays



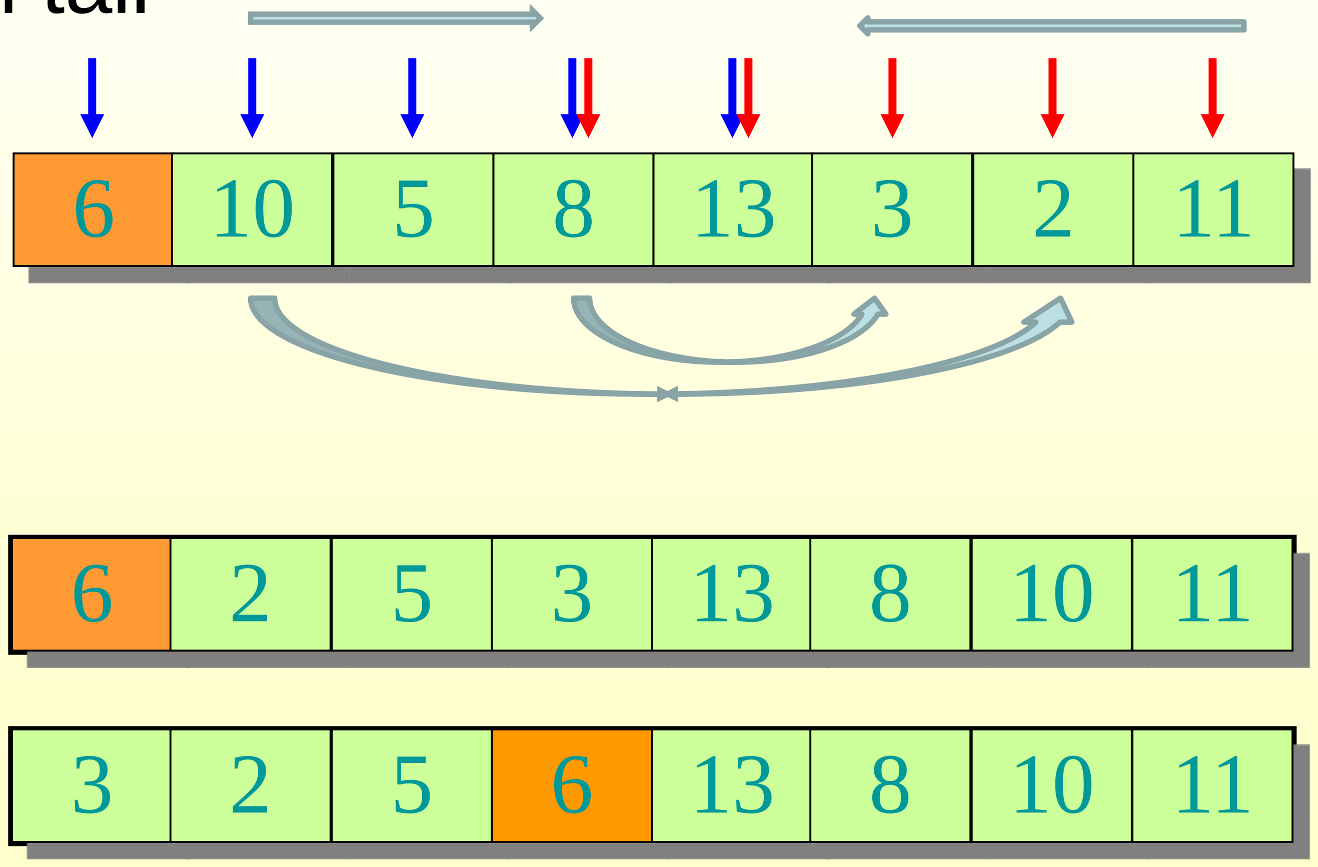
Idea of Partition

- If we are allowed to use a second array, it would be easy

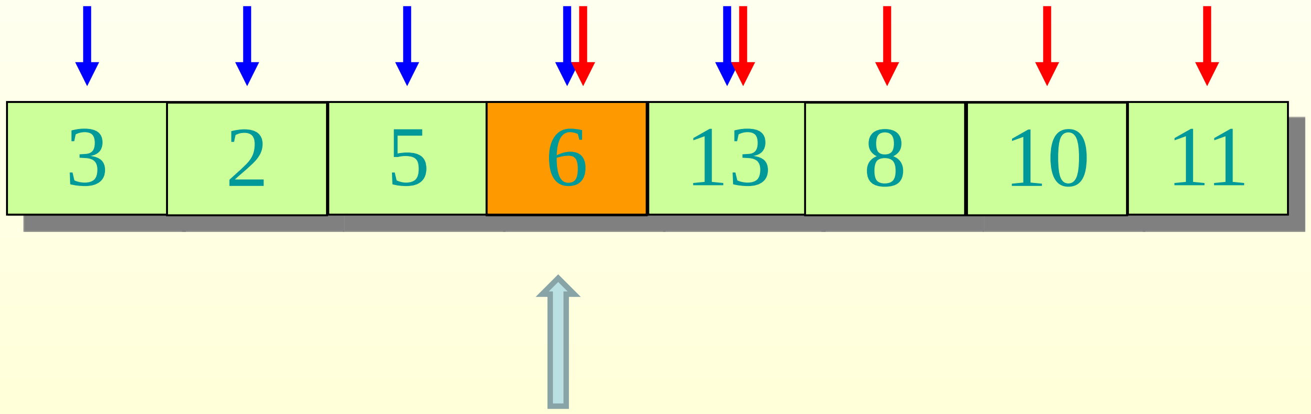


Another Idea: Pairing

- Keep two iterators: one from head, one from tail



In-Place Partition



Pivot element

The “partitioner”

Originally the leftmost

Partition in English

- Partition(A, p, r):
  - Select an element to act as the "pivot" (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] > pivot
  - Decrement j until A[j] < pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Swap A[j] and A[p]
  - Return j

Note: different from book's **`partition()`**, which uses two iterators that both move forward.

Partition in Code

```
Partition(A, p, r)
    x = A[p];          // pivot is the leftmost element
    i = p;
    j = r + 1;
    while (TRUE) {
         repeat
             i++;
        until A[i] > x or i >= j;
        repeat
             j--;
        until A[j] < x or j < i;
        if (i < j)
             Swap (A[i], A[j]);
        else
             break;
     }
     swap (A[p], A[j]);
     return j;
```

What is the running time of ***`partition()`****?*

`partition()` *runs in* $\Theta(n)$ *time*

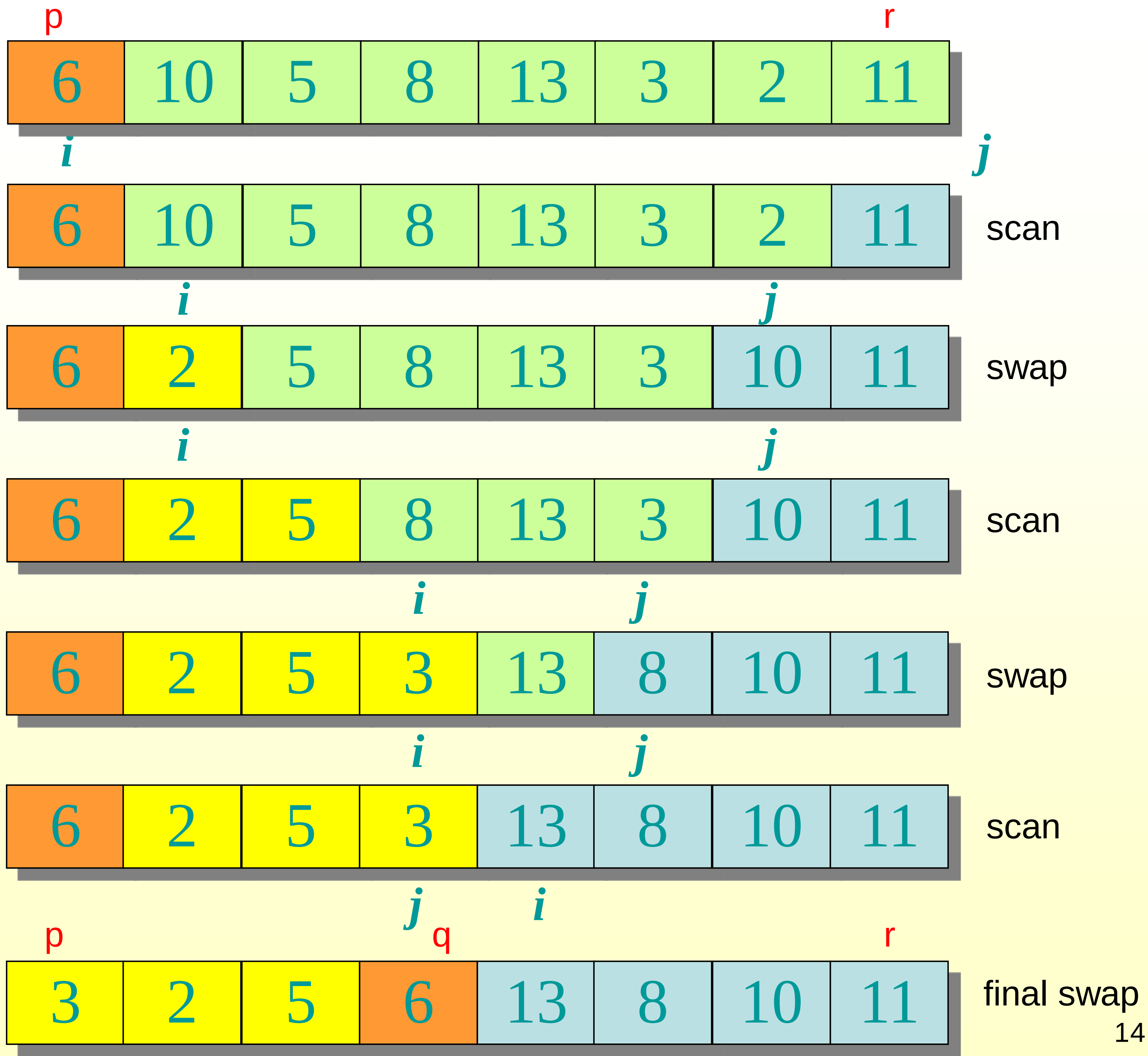
The QuickSort Algorithm

```
QUICKSORT($A$, $p$, $r$)
    if $p < r$
        then $q \leftarrow$ PARTITION($A$, $p$, $r$)
            QUICKSORT($A$, $p$, $q-1$)
            QUICKSORT($A$, $q+1$, $r$)
```

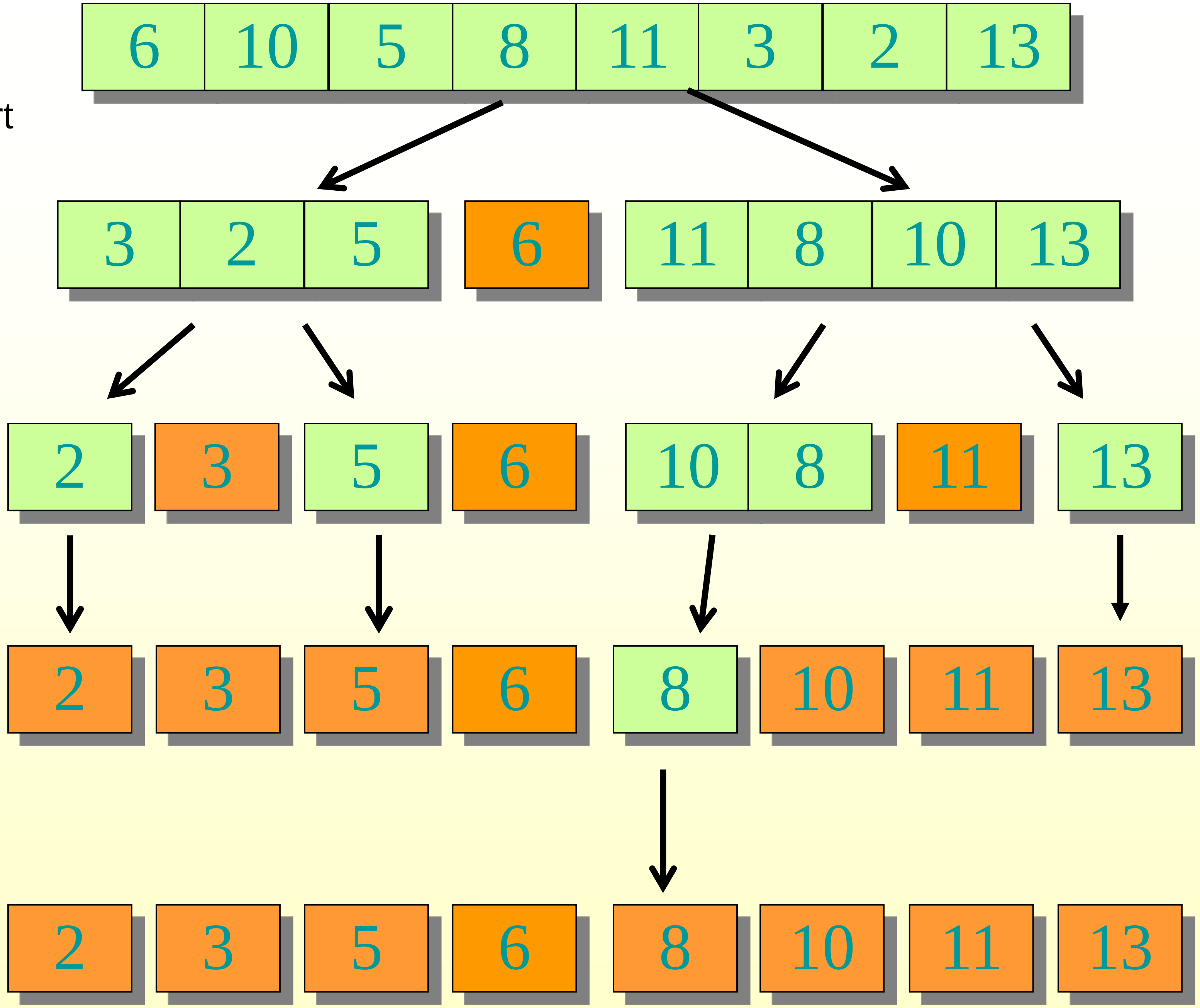
Correctness is quite clear

Pivot
x = 6

Partition example



QuickSort
example



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

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

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

Worst-Case Recursion Tree

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

Worst-Case Recursion Tree

$$T(n) = T(0) + T(n{-}1) + n$$

$T(n)$

Worst-Case Recursion Tree

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

$$\begin{tikzcd}
 & n \arrow[dl, no head] \arrow[dr, no head] & \\
T(0) & & T(n-1)
\end{tikzcd}$$

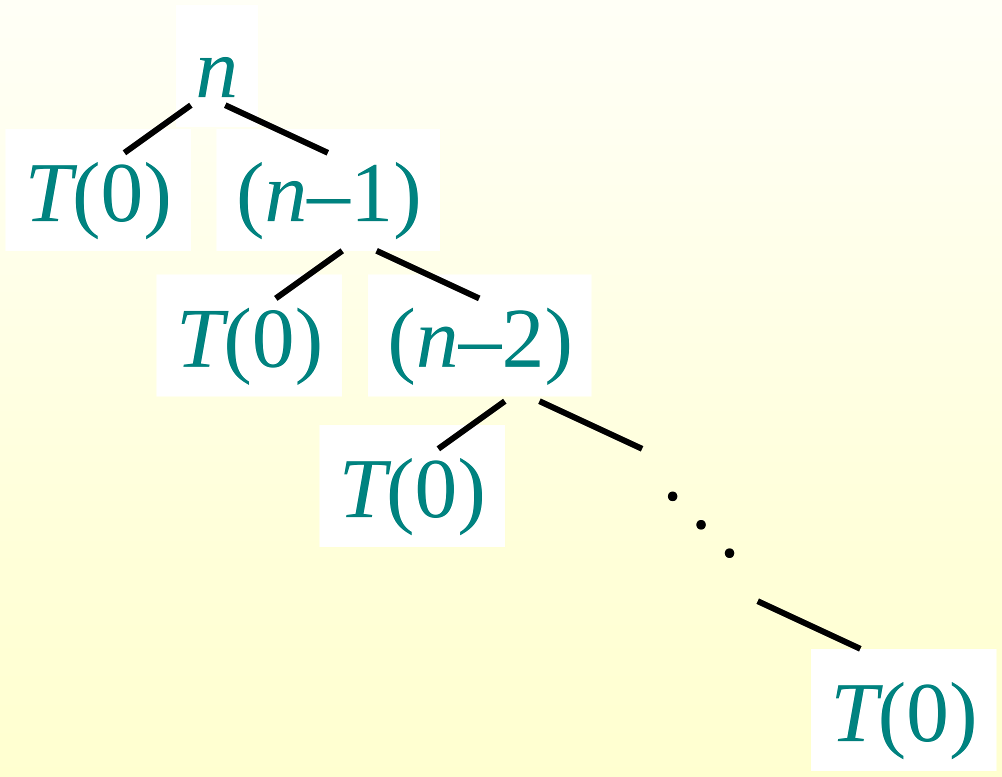
Worst-Case Recursion Tree

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

$$\begin{tikzcd}
& n \arrow[dl, no head] \arrow[dr, no head] & & \\
T(0) & & (n-1) \arrow[dl, no head] \arrow[dr, no head] & \\
& T(0) & & T(n-2)
\end{tikzcd}$$

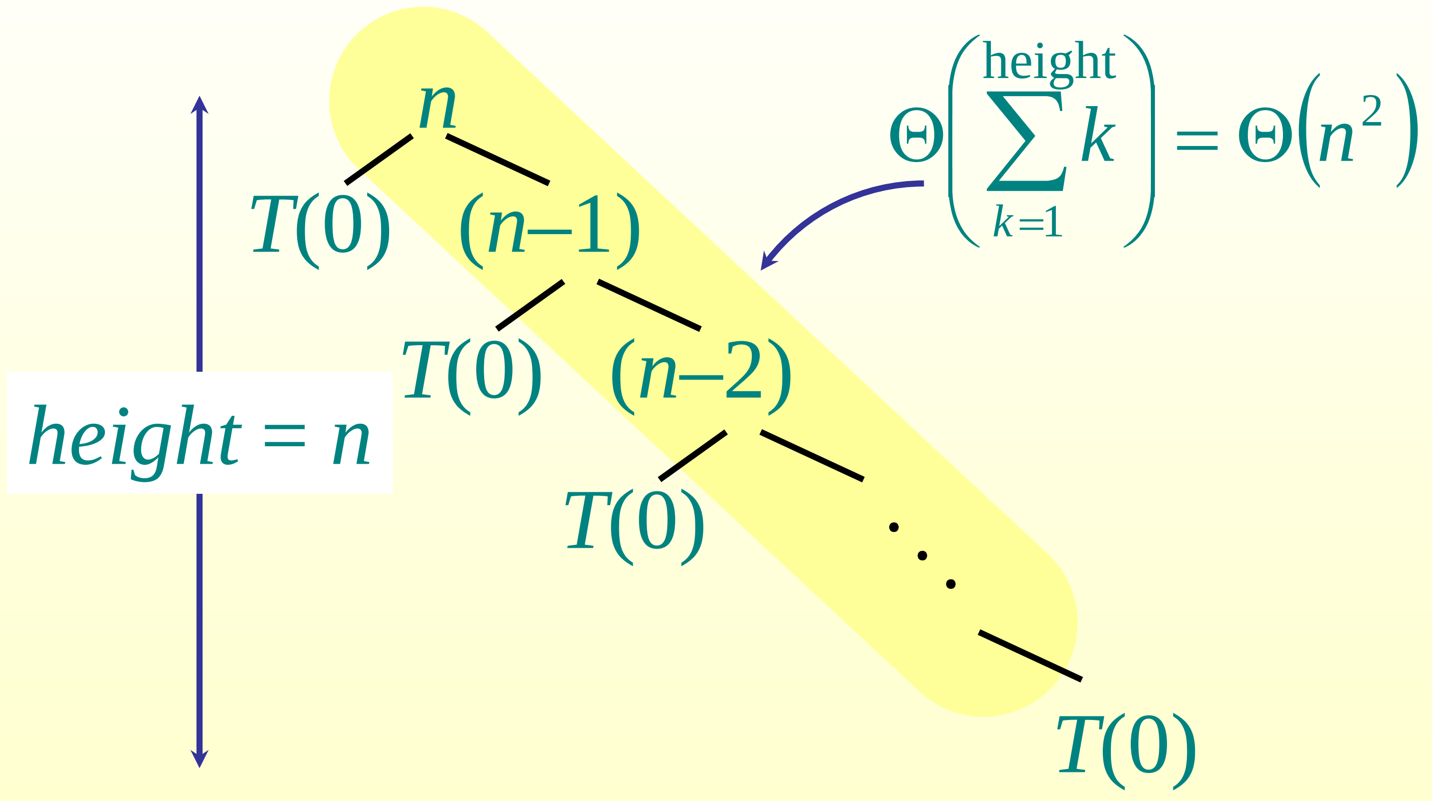
Worst-Case Recursion Tree

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



Worst-Case Recursion Tree

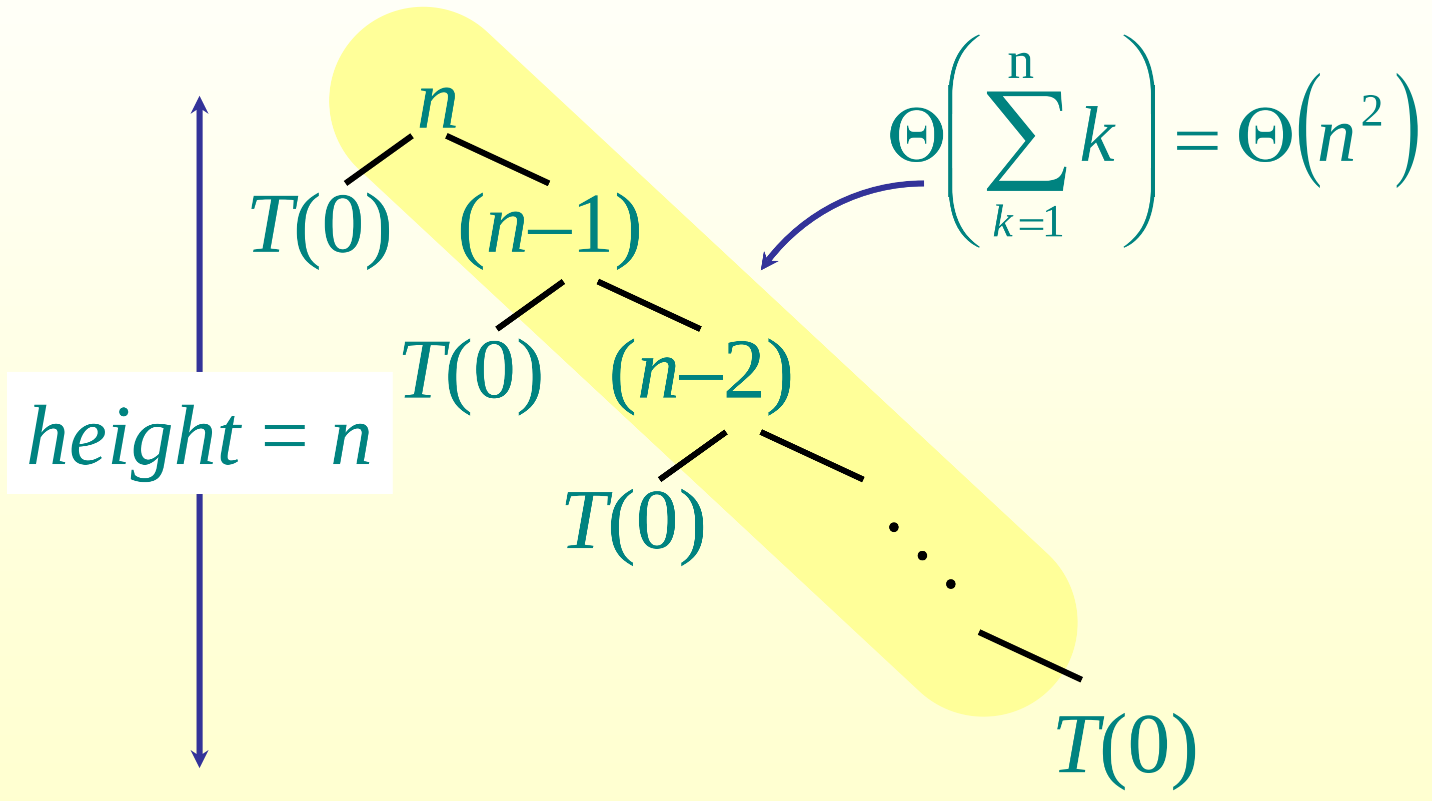
$$T(n) = T(0) + T(n-1) + n$$



n
$T(0)$
$(n-1)$
$T(0)$
$(n-2)$
$T(0)$
$T(0)$
$height = n$
$\Theta\left(\sum_{k=1}^{\text{height}} k\right) = \Theta(n^2)$

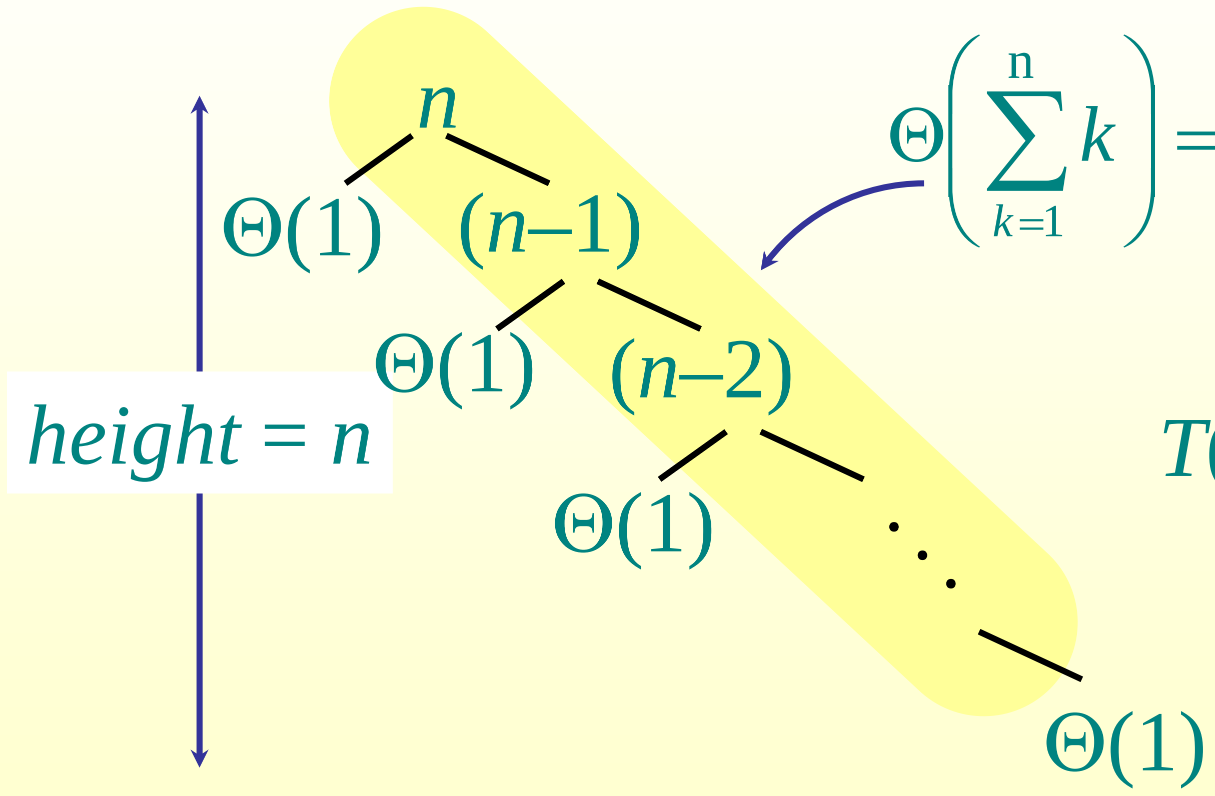
Worst-Case Recursion Tree

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



Worst-Case Recursion Tree

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



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

Best-Case Analysis

(For intuition only!)

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

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

(same as MergeSort)

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

$T(n)$

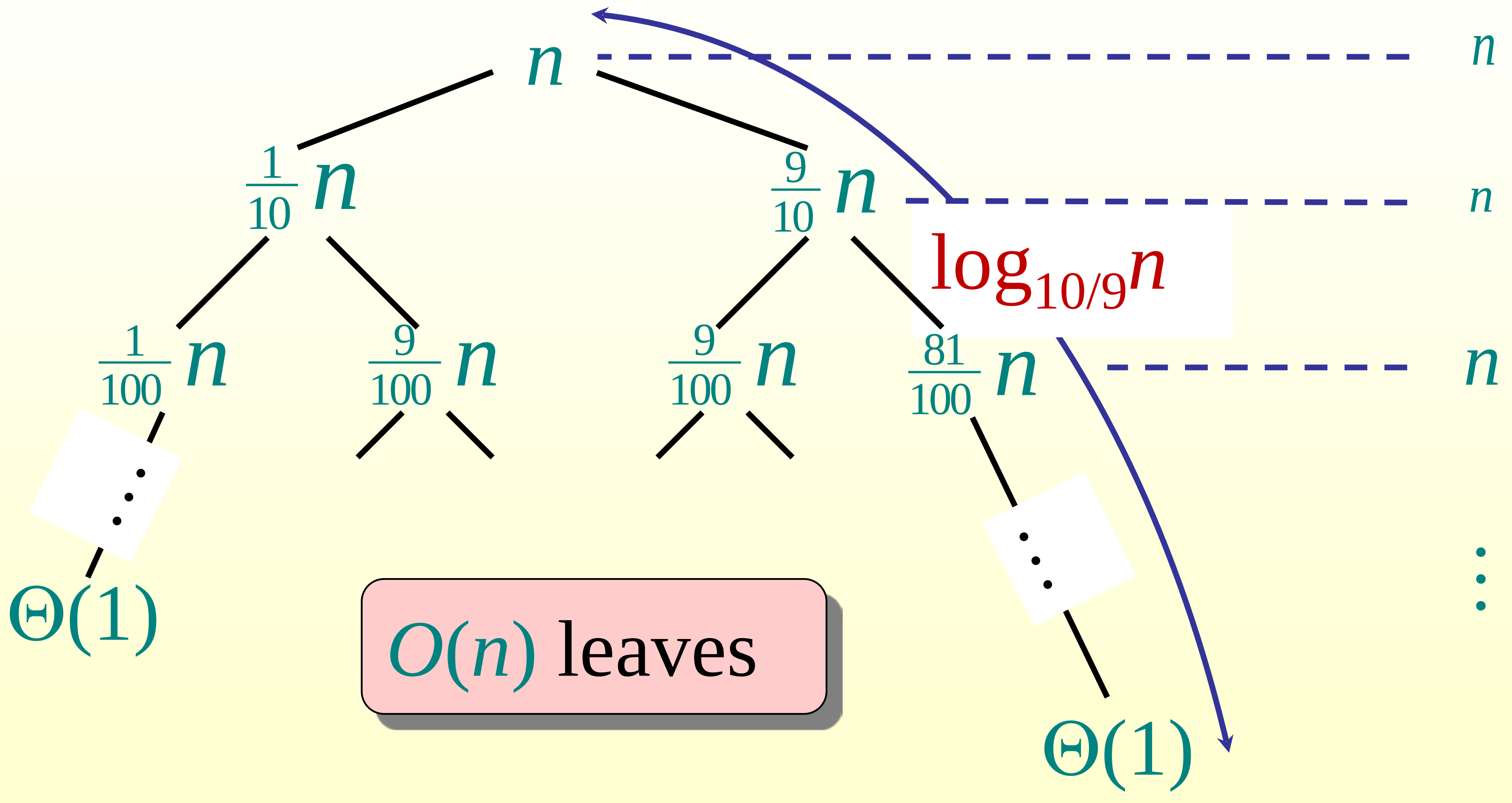
Analysis of “Almost-Best” Case

$$\begin{tikzcd} & n \arrow[dl, dash] \arrow[dr, dash] & \\ T\left(\frac{1}{10}n\right) & & T\left(\frac{9}{10}n\right) \end{tikzcd}$$

Analysis of “Almost-Best” Case

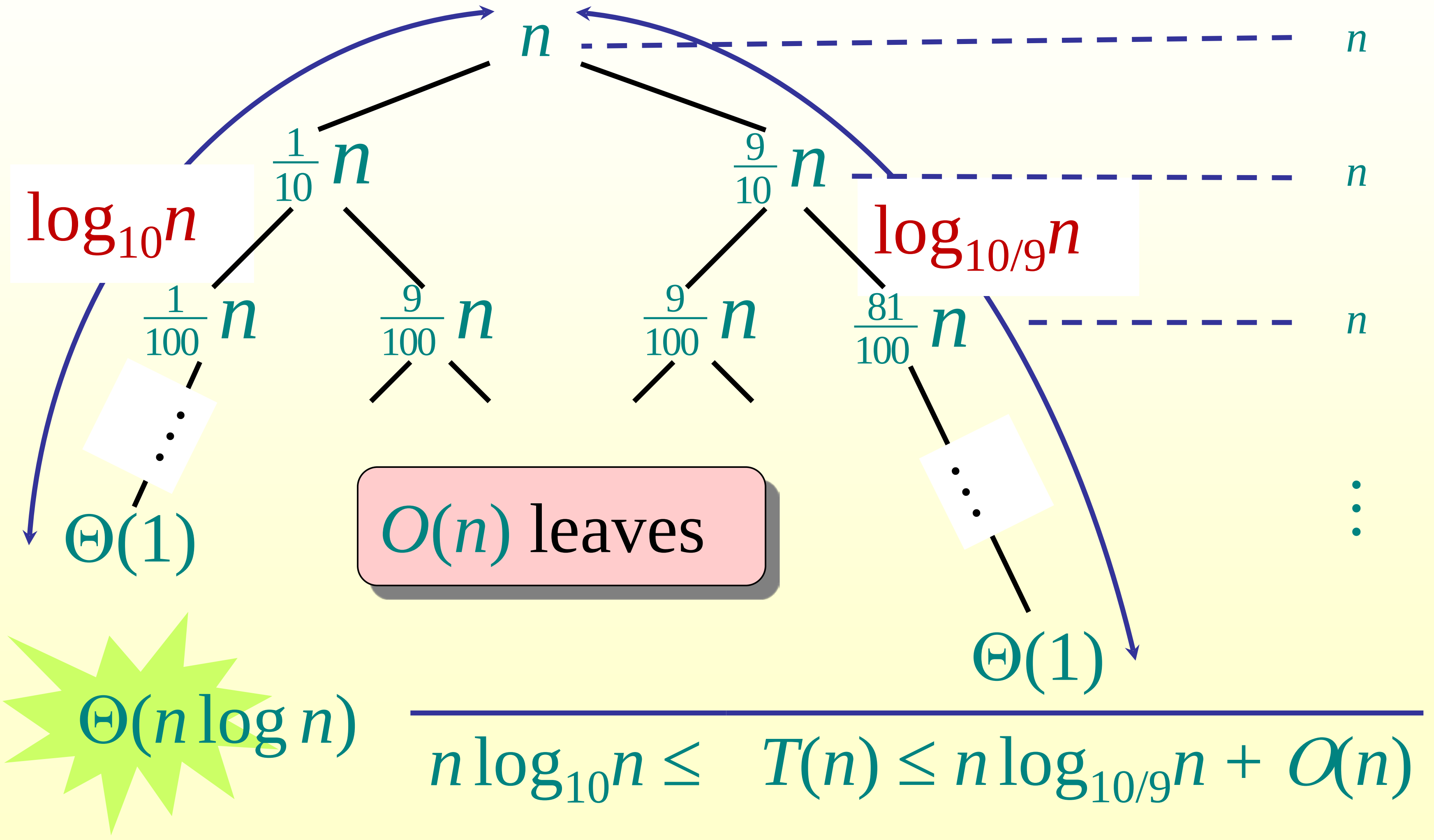
$$\begin{tikzcd}[column sep=small]
 & & & n \arrow[dll, dash] \arrow[drr, dash] & & & \\
 & \frac{1}{10}n \arrow[dl, dash] \arrow[dr, dash] & & & & \frac{9}{10}n \arrow[dl, dash] \arrow[dr, dash] & \\
T\left(\frac{1}{100}n\right) & & T\left(\frac{9}{100}n\right) & & T\left(\frac{9}{100}n\right) & & T\left(\frac{81}{100}n\right)
\end{tikzcd}$$

Analysis of “Almost-Best” Case



$O(n)$ leaves

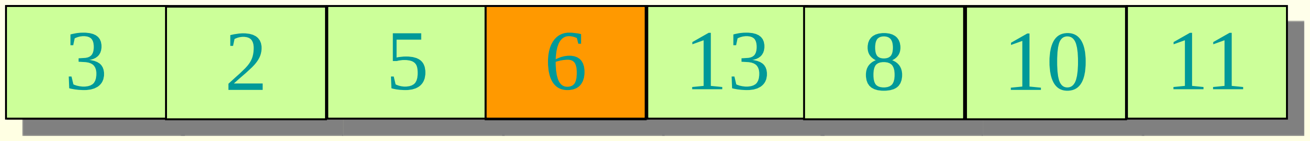
Analysis of “Almost-Best” Case



$$n \log_{10} n \le \quad T(n) \le n \log_{10/9} n + O(n)$$

The Almost-Best Case is Common

$$T(n) = T\left(\frac{1}{1000}n\right) + T\left(\frac{999}{1000}n\right) + \Theta(n)$$



Many more “middle” elements than “extreme” elements

QuickSort Runtimes

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

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

Randomized QuickSort

- Randomly choose an element as pivot
    - Every time need to do a partition, throw a die to decide which element to use as the pivot
    - Each element has 1/n probability to be selected

```
Rand-Partition(A, p, r)
    d = random();   // a random number between 0 and 1
    index = p + floor((r-p+1) * d);  // p <= index <= r
    swap(A[p], A[index]);
    Partition(A, p, r);  // now do partition using A[p] as pivot
```

Running Time of Randomized QuickSort

$$T(n) = \begin{cases} T(0) + T(n-1) + dn & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + dn & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + dn & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

- The expected running time is an average over all cases

In expectation $\longrightarrow$ $$\overline{T}(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left(\overline{T}(k) + \overline{T}(n-k-1)\right) + dn$$

$$\overline{T}(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left(\overline{T}(k) + \overline{T}(n-k-1)\right) + n$$

$$= \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k) + n$$

Say $d = 1$

Solving the Recurrence

1. Recursion tree (iteration) method
   - Good for guessing an answer
2. Substitution method
   - Generic method, rigid, but may be hard
3. Master method
   - Easy to learn, useful in limited cases only
   - Some tricks may help in other cases

Substitution Method

The most general method to solve a recurrence (prove O and Ω separately):

1. ***Guess*** the form of the solution:
   (e.g. using recursion trees, or expansion)
2. ***Verify*** by induction (inductive step).

Expected Running Time of QuickSort (Method 1)

$$\overline{T}(n) = \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k) + n$$

- Guess $\overline{T}(n) = O(n\log n)$
- We have to prove $\overline{T}(n) \le cn\log n$ for some c and sufficiently large n
- Use $T(n)$ instead of $\overline{T}(n)$ for convenience

- Fact: $$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + n$$
- Need to Prove: *T(n) ≤ c n log (n)*
- Assumption: *T(k) ≤ ck log (k) for 0 ≤ k ≤ n-1*
- Prove using substitution method

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + n$$

$$\le \frac{2c}{n}\sum_{k=0}^{n-1} k\log k + n$$

$$\le \frac{2c}{n}\left(\frac{n^2}{2}\log n - \frac{n^2}{8}\right) + n$$ using the fact that $\sum_{k=0}^{n-1} k\log k \le \frac{n^2}{2}\log n - \frac{n^2}{8}$

$$\le cn\log n - \frac{cn}{4} + n$$

$$\le cn\log n$$ if c ≥ 4

log here means lg

Tightly Bounding The Key Summation

$$\sum_{k=0}^{n-1} k \lg k = \sum_{k=1}^{n-1} k \lg k$$

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

Split the summation for a tighter bound

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n$$

The* lg *k in the second term is bounded by* lg *n

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move the* lg *n outside the summation

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The* lg *k in the first term is bounded by* lg *n/2

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k(\lg n - 1) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

lg *n/2* = lg *n* - 1

$$= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move (*lg *n* - 1*) outside the summation

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

What are we doing here?

$$= \lg n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Distribute the (lg n - 1)

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summations overlap in range; combine them

$$= \lg n \left(\frac{(n-1)n}{2} \right) - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \left(\frac{(n-1)n}{2}\right)\lg n - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summation bound so far

$$\le \frac{1}{2}\left[n(n-1)\right]\lg n - \sum_{k=1}^{n/2-1} k$$

Rearrange first term, place upper bound on second

$$\le \frac{1}{2}\left[n(n-1)\right]\lg n - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$$

$$\le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

Multiply it all out

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

$$= \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 + \left(\frac{n}{4} - \frac{n}{2}\lg n\right)$$

$$\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \text{ when } n \ge 2$$

Done!

Expected Running Time of QuickSort (Method 2)

If X denotes the number of comparisons performed in the PARTITION subroutine over the entire execution of QUICKSORT on an n-element array, then the running time of QUICKSORT is $O(n + X)$.

We need to find X, the total number of comparisons performed over all calls to PARTITION.

Partition in English

- Partition(A, p, r):
  - Select an element to act as the "pivot" (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] > pivot
  - Decrement j until A[j] < pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Swap A[j] and A[p]
  - Return j

Note: different from book's **`partition()`**, which uses two iterators that both move forward.

A Probabilistic Formulation

QUICKSORT: Performance – Expected RunTime

1. Rename the elements of A as $z_1, z_2, \ldots, z_n$, so that z_i is the i^{th} smallest element of A (sorted order).
2. Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$.
3. Question: how often does the algorithm compare z_i and z_j?
4. Answer: at most once – notice that all elements in every (sub)array are compared to the pivot once, and will never be compared to the pivot again (since the pivot is removed from the recursion).
5. Define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$, the indicator variable of this event. Comparisons are over the full run of the algorithm.

Calculating with Expectations

6. Since each pair is compared at most once, we can write

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

7. Taking expectations of both sides:

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}.$$

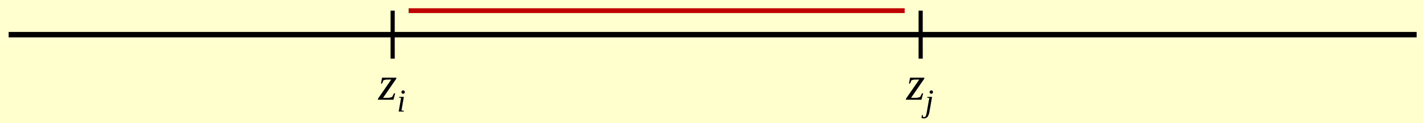
8. We need to compute $\Pr\{z_i \text{ is compared to } z_j\}$.

9. We will assume all z_i and z_j are distinct.

10. For any pair z_i, z_j, once a pivot x is chosen so that $z_i < x < z_j$, z_i and z_j will never be compared again (why?).

The Key Probability

11. If z_i is chosen as a pivot before any other item in Z_{ij}, then z_i will be compared to every **other** item in Z_{ij}.
12. Same for z_j.
13. z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j.
14. What is that probability? Until a point of Z_{ij} is chosen as a pivot, the whole of Z_{ij} is in the same partition, so every element of Z_{ij} is equally likely to be the first one chosen as a pivot.



The Key Probability

15. Because Z_{ij} has $j - i + 1$ elements, and because pivots are chosen randomly and independently, the probability that any given element is the first one chosen as a pivot is $1/(j-i+1)$. It follows that:

16. $\Pr\{z_i \text{ is compared to } z_j\}$

$= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}$

$= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} +$

$\Pr\{ z_j \text{ is first pivot chosen from } Z_{ij}\}$

$= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1).$

Harmonic Numbers to the Rescue

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E\left[X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}. \qquad (7)$$

17.Replacing the right-hand-side in 7, and grinding through some algebra:

$$E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} 2H_n = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n).$$

And the result follows.

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Expected Running Time of QuickSort (Method 3)

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + n$$

$$nT(n) = 2\sum_{k=0}^{n-1} T(k) + n^2$$

$$(n-1)T(n-1) = 2\sum_{k=0}^{n-2} T(k) + (n-1)^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

$$nT(n) = (n+1)T(n-1) + 2n - 1$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{(n+1)n}$$

$$S(n) = \frac{T(n)}{n+1}$$

$$S(n) = S(n-1) + \frac{2n-1}{(n+1)n} \approx S(n-1) + \frac{2}{n+1}$$

$$S(n) = 2\sum_{k=1}^{n} \frac{1}{k+1} \approx H_n = O(\log n)$$

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

Picking the Pivot

- Use the first element as pivot
  - if the input is random, OK
  - if the input is presorted (or in reverse order)
    - all the elements go into only one of the subfiles
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

Picking the Pivot

- Use the median of the array
  - Partitioning always cuts the array into roughly half
  - However, hard to find the exact median
    - e.g., sort an array to pick the value in the middle
  - Possible in Θ(n) time, as we will later in the course [but complicated]
  - An optimal quicksort (O(N log N))

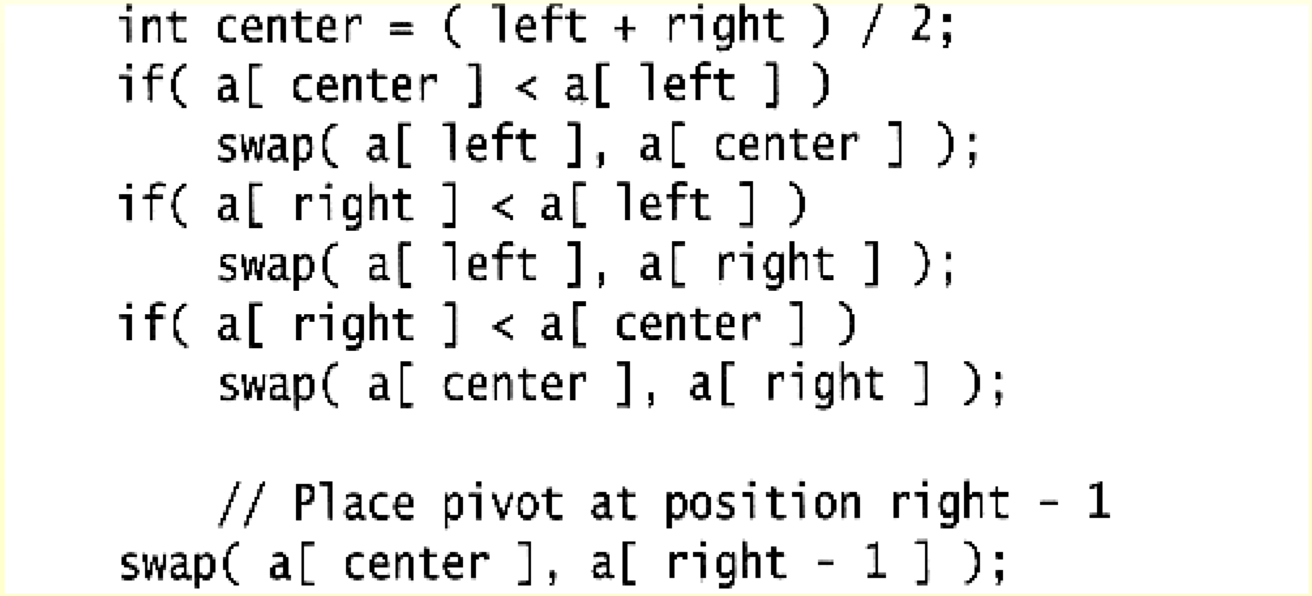
Pivot: Median of Three

- We will use median of three
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that

A[left]	=	Smallest
A[right]	=	Largest
A[center]	=	Median of three

  - Pick A[center] as the pivot
  - Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

median3



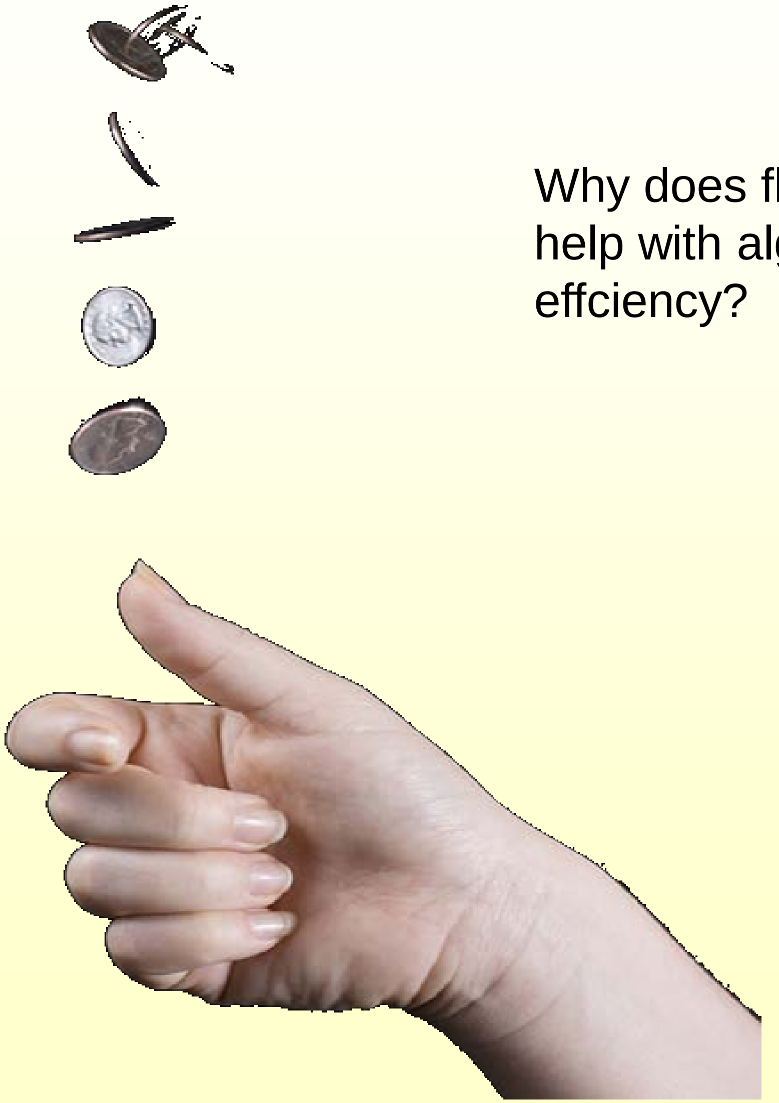
```
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );

        // Place pivot at position right - 1
    swap( a[ center ], a[ right - 1 ] );
```

Stability of Sorting

- Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values).
    - That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

- MergeSort and InsertionSort can be implemented to be stable. QuickSort in a normal implementation is not.
- However, any comparative sorting algorithm can be made stable. [How?]

Randomized Algorithms



Why does flipping coins help with algorithm effciency?