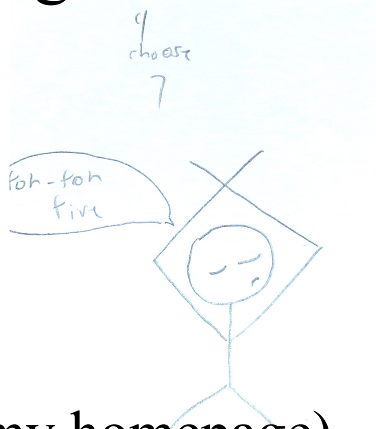
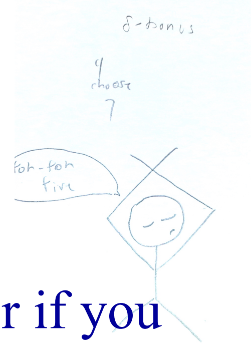
CS545 – Design and Analysis of Algorithms



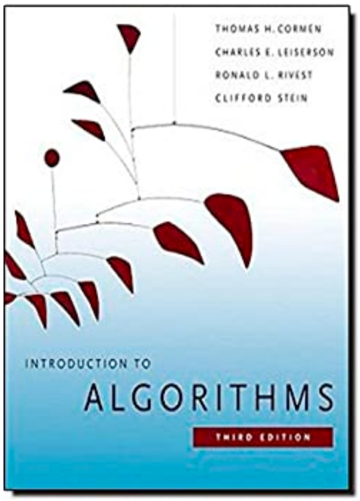
- **Webpages**
- Course webpage – google doc (reach via my homepage)
- Use D2L to reach recordings of lectures (Panopto), calendar
- Use Gradescope to submit his and view feedback
- Use Piazza for course communication, discussions and announcements.
- Use Overleaf to view assignments.

CSc545 - Regulation, Bureaucracy

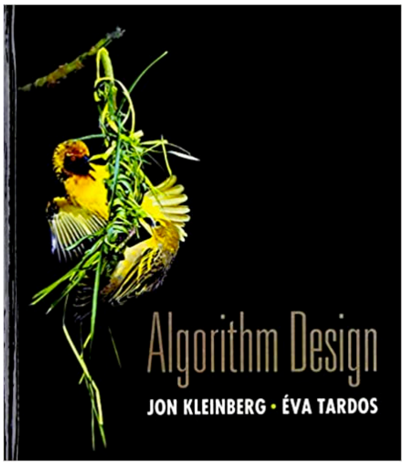


1. Video recording
2. Web Resources
3. Prerequisites (course is mostly self contained, but harder if you did not pass cs345.
4. Piazza.
   I. Post are for clarifications.
   II. Be careful not to share any hints in your posts
   Eg. "*are we allowed to use Quicksort for the solution of hw3 Q7*" is a violation of code of conduct, considered cheating, and could get you blocked from piazza.
   I. If you have any doubts, send a private message.
5. Attendance - strongly recommended.
   1. Active learning - your webcam should be **on** during active learning (talk to me if there are any technical difficulties).

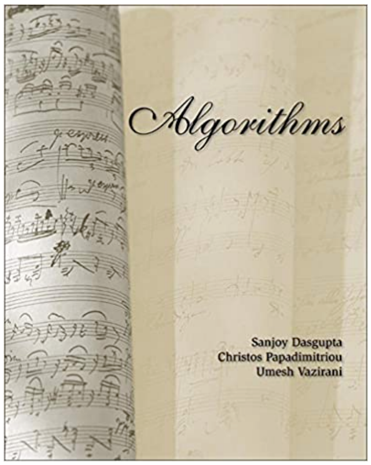
1. Textbook



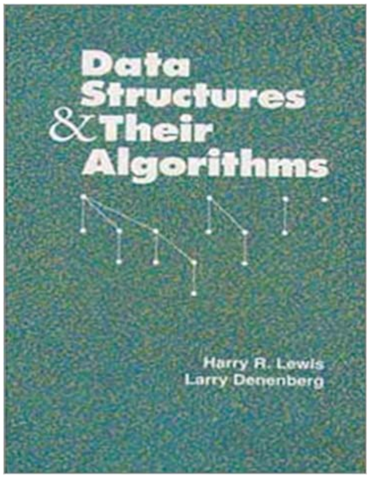
CLRS



Kleinberg & Tardos



Sanjoy Dasgupta



Lewis Denenberg

Course slides

Assignments in Exams

- About 6 or 7 homeworks.
- All theoretical (no programming)
- Grade of lowest one is dropped
- Possibly some of them could be submitted in pairs

- One midterm. One Final exam

Homeworks workflow. Collaboration vs Cheating

- Alg: Once a homeworks is published
    - ❑ Read questions
        - ❑ If needed, re-watch lectures (Alon and Others) online,
    - ❑ Thinks really hard. Discover what does **not** work and why
    - ❑ Meet your peers and discuss and does/does not work and why?
    - ❑ Write Solutions yourself.
    - Diverging from this algorithm might improve your hw grade but is likely to impact your exams grades (not to mention ethical issues, honor code etc).
- Homework's rules.
    - Collaborations ++. Brainstorming in **small** groups
    - Give credit. Specify your contribution to each solution (in %).
    - Sharing text is cheating.

Introduction to Algorithms

- In this course, we will discuss problems, and algorithms for solving these problems.

- There are so many algorithms – why focus on the ones in the syllabus ?

Why study algorithms and performance?

Why study algorithms and performance?

- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
  - (e.g., by using big-*O* –notation.
  - Will see lots of `big-*O'**s* of quantities you might have not seen before:
    - (CPU, Space, I/O, parallel steps, GPU)
- In real life, many algorithms, though different from each other, fall into one of several ***paradigms*** (discussed shortly).
- These paradigms can be studied, and applied for new problems

Why these algorithms (cont.)

1. **Main paradigms:**
   a) Greedy algorithms
   b) Divide-and-Conquers
   c) Dynamic programming
   d) Brach-and-Bound (mostly in AI)
   e) Etc etc.

2. **Other reasons:**
   a) Relevance to many areas:
      - E.g., networking, internet, search engines…
   b) Coolness

Other goals of the course

- Knowing when running time counts, and what to do when it does

- Magic of randomness and sampling

O-notation

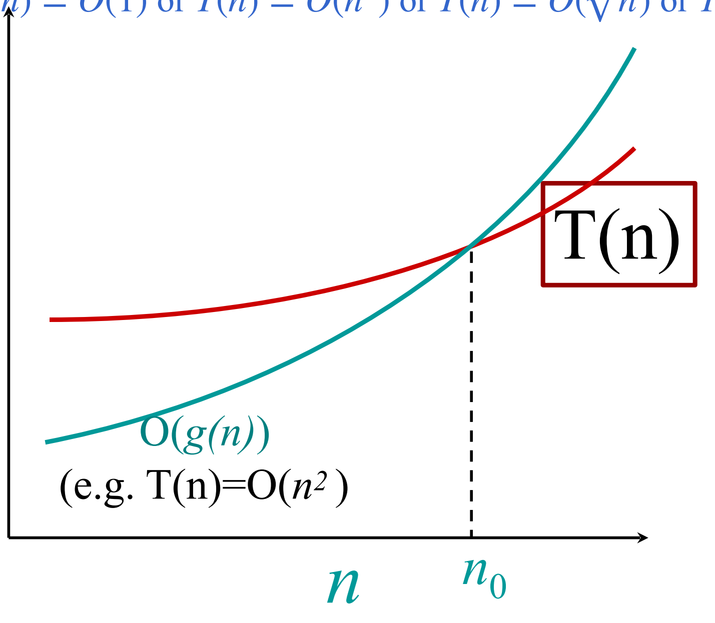
we say that $T(n) = O(g(n))$ **iff**

there exists positive constants c_1, and n_0 such that

$0 \leq T(n) \leq c_1 g(n)$ for all $n \geq n_0$

Common examples. We would say that the running time T(n), on an input of size n,

Is $T(n) = O(1)$ or $T(n) = O(n^2)$ or $T(n) = O(\sqrt{n})$ or $T(n) = 2^{2^{2\cdots}}$ n times

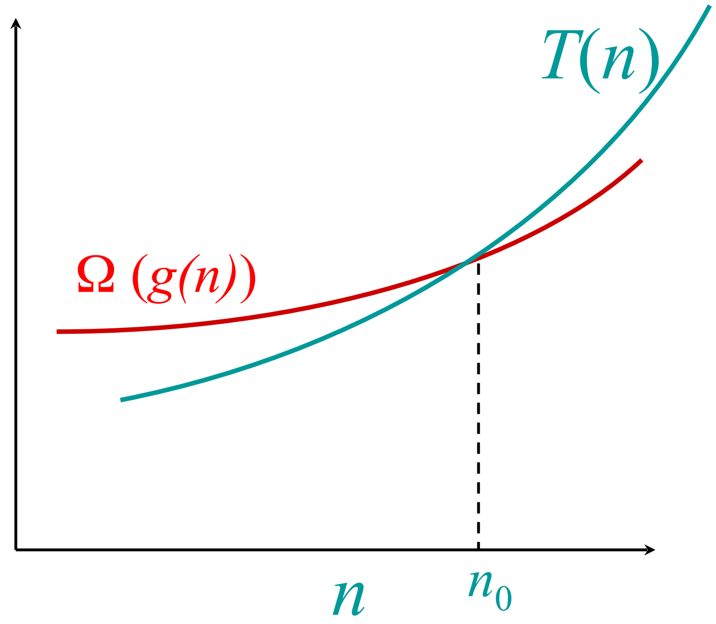


- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- **Asymptotic analysis** is a useful tool to help to structure our thinking.

Ω-notation

Math:

We say that $T(n) = \Omega(g(n))$ **iff**
there exists positive constants c_2, and n_0
such that
$0 \le c_1 g(n) \le T(n)$ for all $n \ge n_0$



Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Omega\ (n^3)$

Θ-notation

We say that $T(n) = \Theta(g(n))$ **iff**

there are positive constants

c_1, c_2, and n_0

such that

$$0 \leq c_1 g(n) \leq T(n) \leq c_1 g(n)$$

for every n, provide that $n \geq n_0$

in other words, we could say that

$T(n) = \Theta(g(n))$

iff it is true that

$T(n)=O(g(n))$ and that $T(n) = \Omega(g(n))$.

For example, for every size *n* of an input array, bubble sort, insertion sort and swap-sort will never needs more than n^2 operations (up to a constant).

So their running time is $O(n^2)$.

On the other hand, we can find an input (one is enough) that causes their running time to be no less than n^2. So their running time is also $\Omega(n^2)$.

Putting it together, their running time is $\Theta(n^2)$

Notation - cont

So if $T(n) = O(n^2)$ then we are also sure that
$T(n) = O(n^3)$ and that
$T(n) = O(n^{3.5})$ and
$T(n) = O(2^n)$

But it might or might not be true that $T(n) = O(n^{1.5})$.

However, if $T(n) = \Omega(n^2)$ then it is **not** true that
$T(n) = O(n^{1.5})$

Big difference between O and Ω: we can talk about Ω of a **problem**
(that is, any algorithm that solves this problem
takes Ω(something)

Eg. Sorting takes $\Omega(n \log n)$

Sometimes, the lower bound refers to for any algorithm

Famous examples:

Sorting: Any algorithm that sort n real numbers takes $\Omega(n \log n)$ time in the worst (slowest) case.

Element uniqueness: Given an array of n keys, determine if there is any key that appears more than once. (just a yes/no answer)

How could we solve this problem?

Sometimes, the lower bound refers to for any algorithm

Famous examples:

Sorting: Any algorithm that sort n real numbers takes $\Omega(n \log n)$ time in the worst (slowest) case.

Element uniqueness: Given an array of n keys, determine if there is any key that appears more than once. (just a yes/no answer)

How could we solve this problem?
Sorting - takes O(n log n) ? Can we do better ?

Turn out that in the general case, no.
Element uniqueness takes $\Omega(n \log n)$ time in the worst case.

Exampels

```
NoNeed(n){
    •   If (n<1) return ;
    •   Print(‘*’)
    •   NoNeed(n-1)
}
```

Recursion formula: $T(n)=c+T(n-1),$ where $T(1)=c.$ We can solve it using the **iteration method:**

$T(n)= \ c+T(n-1)=$

$c+\{c+T(n-2)\} = \quad 2c+T(n-2) =$

$2c+\{ c+T(n-3) \} = 3c+T(n-3) =\ldots =$ (pick $k<n$)

$kc+T(n-k) =$ (setting $k = n-1$) …

$(n-1)c+T(1)=nc$

Example 2

```
NoNeed($n$){
    if ($n<1$) return ;
    for( $i=1$ ; $i<n$ ; $i$++)   print(*)
    NoNeed($n-1$)
}
```

Recursion formula: $T(n)=cn+T(n-1)$, where $T(1)=c$. We can solve it using the **iteration method:**

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

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

$c[n+(n-1)]+\{c(n-2)+T(n-3)\}$

$=c[(n)+(n-1)+(n-2)+(n-3)]+T(n-4) =\ldots =$ (pick $k<n$)

$=c[(n)+(n-1)+(n-2)+ (n-3)+\ldots+(n-k)]+T(n-k-1) =$

(setting $k = n-1$) …

$c[n+ n-1 + n-2 + n-3 +\ldots+1]+T(1)=$

$c[1+2+3+\ldots +n]+T(1)= cn(n+1)/2 +c= \Theta(n^2)$.

We are using the formula for arithmetic sum 1+2+3+..n=n(n+1)/2

Examples 3

```
1. Read(n);
2. $k=1$ ;
3. While( $k \le n$ )
4.   {   $k=2k$ ;  }
```

What is the running time of this code (as a function of n) ?

- We know that each iteration takes *O(1)* times. Need to find the number time line 3 is executed.
  - After the first iteration $k=2=2^1$
  - After the 2nd iteration $k=4=2^2$
  - After the 3rd iteration $k=8=2^3$
  - ….
  - After the i'th iteration $k=2^i$

Cheatsheet :

- $\log(ab)=\log(a)+\log(b)$
- $\log(a^b) = b \log a$
- $\log_a(x) = \log_b(x) / \log_b a$
- $x \le y$ implies $\log_2(x) \le \log_2(y)$

Lets count the number **j** of times that the condition of line 3 was checked and yield true.

- If the condition is true, then $k \le n$. But $k = 2^j$ So $2^j \le n$.
- Taking $\log_2$ from both sides, we have that

  $\log_2 k = \log_2(2^j) \le \log_2(n)$ or..

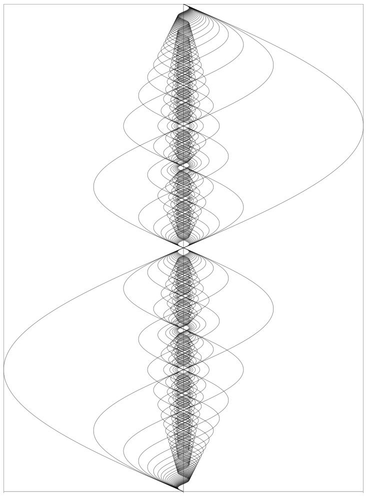
$$\log_2(k) = \log_2(2^j) = j\log_2(2) = j \cdot 1 \le \log_2(n) \text{ ….. Or}$$

$j=O(\log_2 n)$. $T(n)=O(\log n)$

-

Examples 4

```
read(n) ;
for(i=1 ; i < n ;  i++)
  for( j=i ; j <n ; j += i )
  print( "*" ) ;
```

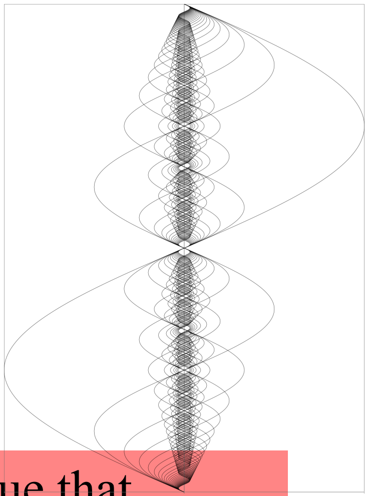


- Time Complexity Analysis – first approach:
  - The outer loop (on *i*) runs exactly *n-1* times
  - The inner loop (on *j*) runs O(*n*) times.
  - Together $T(n)=O(n^2)$.

Examples 4

```
read(n) ;
for(i=1 ; i < n ;  i++)
  for( j=i ; j <n ; j += i )
  print( “*” ) ;
```

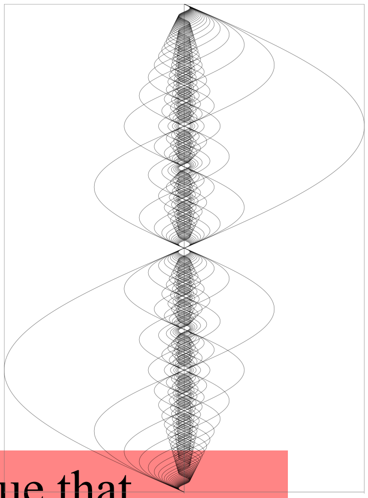
- Time Complexity Analysis – first approach:
  - The outer loop (on *i*) runs exactly *n-1* times
  - The inner loop (on *j*) runs O(*n*) times.
  - Together $T(n)=O(n^2)$.



Is it true that
the running time is $\Omega(n^2)$?

Examples 4

```
read(n) ;
for(i=1 ; i < n ;  i++)
  for( j=i ; j <n ; j += i )
  print( "*" ) ;
```



- Time Complexity Analysis – first approach:
  - The outer loop (on i) runs exactly $n-1$ times
  - The inner loop (on j) runs $O(n)$ times.
  - Together $T(n)=O(n^2)$.

Is it true that the running time is $\Omega(n^2)$?

- More "sensitive" analysis:
  - For $i=1$ we run through $j=1,2,3,4\ldots n$, total n times.
  - For $i=2$ we run through $j=2,4,6,8,10\ldots n$, total $n/2$ times.
  - For $i=3$ we run through $j=3,6,9,12\ldots n$, total $n/3$ times .
  - For $i=4$ we run through $j=4,8,12,16\ldots n$, total $n/4$ times.
  - For $i=n$ we run through $j=n$, total $n/n=1$ times.
- Summing up: $T(n)=n+n/2+n/3+n/4+\ldots n/n =$

$$n(1+1/2+1/3+1/4+\ldots 1/n) \approx n \ln n$$

Harmonic Sum (the image shows A wave and its harmonics, with wavelengths (credit: wikipedia)

Example 5

Pay attention -
very relevant to this course

```
Read(n) ;     a=0.5
While( n>1)  {
    For( j=1; j ≤ n ; j++ )  print("*")  ;
    n=a*n ;
}
```

- The **first** time the outer loop is called, the "print" is called ***n*** times.
- The **2nd** time the outer loop is called, the "print" is called ***an*** times.
- The **3rd** time the outer loop is called, the "print" is called ***a^2n*** times…
- The **k'th** time the outer loop is called, the "print" is called ***$a^k n$*** times

- Let ***t*** be the number of iterations of the outer loop. Then the total time

$$= n + an + a^2n + a^3n + \ldots a^tn = n(1 + a + a^2 + a^3 + \ldots a^t) <$$
$$n(1 + a + a^2 + a^3 + \ldots a^{\,t} + \ldots) = n / (1-a) = O(n).$$

- Same analysis holds for any ***$a<1$***

Recall: $1+a+a^2+\ldots+a^t = (1-a^{\,t+1})/(1-a).$
If $a<1$ **then** $1+a+a^2+\ldots+a^t+\ldots = 1/(1-a)$

Geometric sum

Properties of big-O

- **Claim:** if $T_1(n)=O(g_1(n))$ and $T_2(n)=O(g_2(n))$ then $T_1(n)+T_2(n)=O(g_1(n) + g_2(n))$

- **Example**: $T_1(n)=O(n^2)$, $T_2(n)=O(n \log n)$ then $T_1(n)+T_2(n)=O(n^2 + n \log n) =O(n^2)$

- **Proof:** We know that there are constants n_1, n_2, c_1, c_2 **s.t.**
  - for every $n>n_1$ $T_1(n) < c_1 g_1(n)$. (definition of big-O)
  - for every $n>n_2$ $T_2(n) < c_2 g_2(n)$. (definition of big-O)

  - Now set $n'=\max\{n_1, n_2\}$, and $c'=c_1+c_2$, then
    - for every $n>n'$ we have that
    - $T_1(n)+T_2(n) < c_1 g_1(n) + c_2 g_2(n) \leq$
      $c' g_1(n) + c' g_2(n) =$
      $c'(g_1(n) + g_2(n))$

More properties of big-O

- **Claim:** if $T_1(n)=O(g_1(n))$ and $T_2(n)=O(g_2(n))$ then
  $T_1(n)\ T_2(n)=O(g_1(n)\ g_2(n))$

- **Example:** $T_1(n)=O(n^2)$, $T_2(n)=O(n \log n)$ then

  $T_1(n)\ T_2(n)=O(n^3 \log n)$

- Similar properties hold for Θ, Ω

Quicksort, as an example to randomize algorithms

Goals:

1. Introduction (and hopefully refreshing) of the QS algorithm
2. Introduction to **random variables** and **expectation**
3. Worst case vs. Expected running time
4. Randomized algorithm for Median Selection

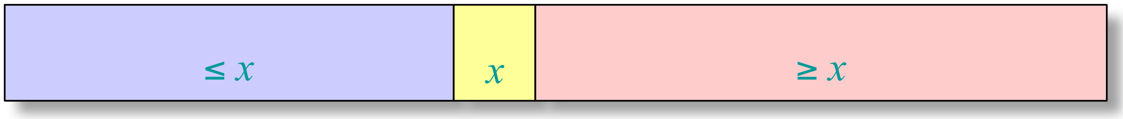
Divide and conquer

Quicksort an n-element array:

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

2. ***Conquer:*** Recursively sort the two subarrays.

- ***Combine:*** Trivial.



Key: *Linear-time partitioning subroutine.*

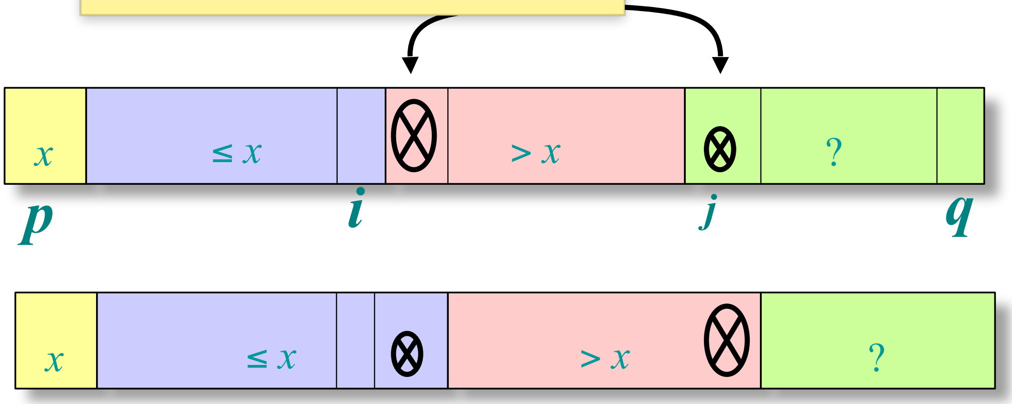
Partitioning subroutine (first attempt, to be improved)

```
PARTITION$(A, p, q)$          ▻ $A[p\,..\,q]$
    —— some improvement will come here (later) —-
    $x \leftarrow A[p]$          ▻ pivot = $A[p]$
    $i \leftarrow p$
    for $j \leftarrow p+1$ to $q$ ▻ j is hunting for small keys
        do if $A[j] \le x$                              the left.
                then{                                   >x
                                                        A[i]>x
                }
    exchange $A[p] \leftrightarrow A[i$
    return $i$
```

It is possible that all keys are >x or all <x

At place i+1 the key is

Invariant:



Partitioning subroutine (first attempt, to be improved)

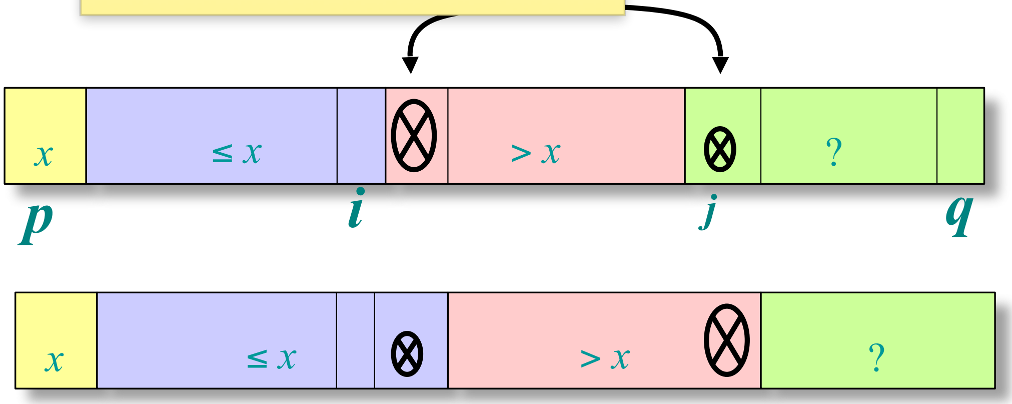
Running time = $O(n)$ for n elements.

```
PARTITION($A, p, q$)          ▻ $A[p\,..\,q]$
    —— some improvement will come here (later) —-
    $x \leftarrow A[p]$                ▻ pivot = $A[p]$
    $i \leftarrow p$
    for $j \leftarrow p + 1$ to $q$ ▻ j is hunting for small keys
         do if $A[j] \leq x$                              the left.
                 then{
                                                          >x
                                                          $A[i]$>x
                 }
    exchange $A[p] \leftrightarrow A[i$
    return $i$
```

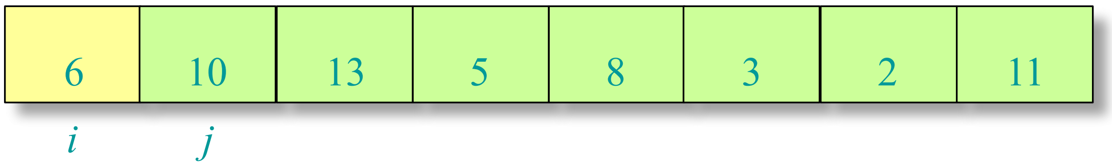
It is possible that all keys are >x or all <x

At place i+1 the key is

Invariant:

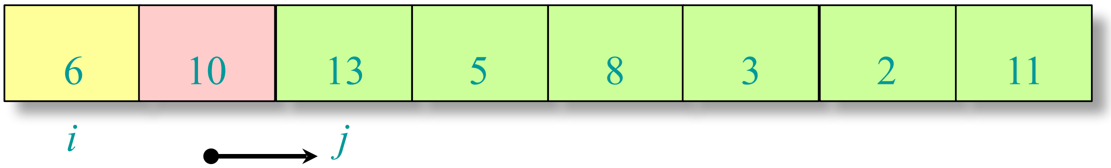


Example of partitioning



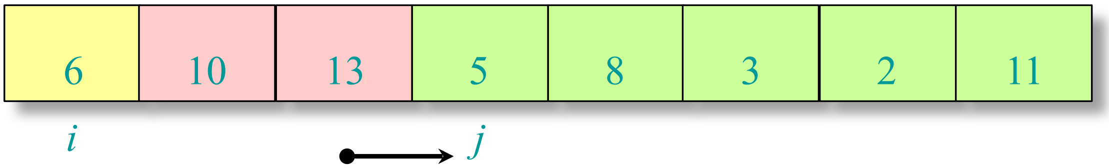
6	10	13	5	8	3	2	11
i	*j*						

Example of partitioning



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

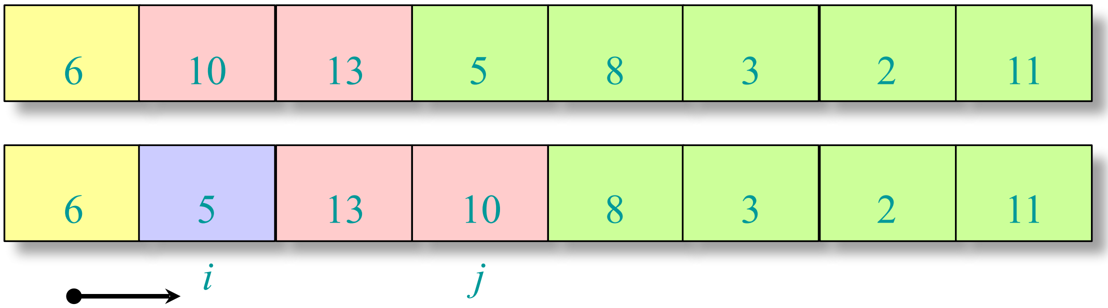
Example of partitioning



6	10	13	5	8	3	2	11

i *j*

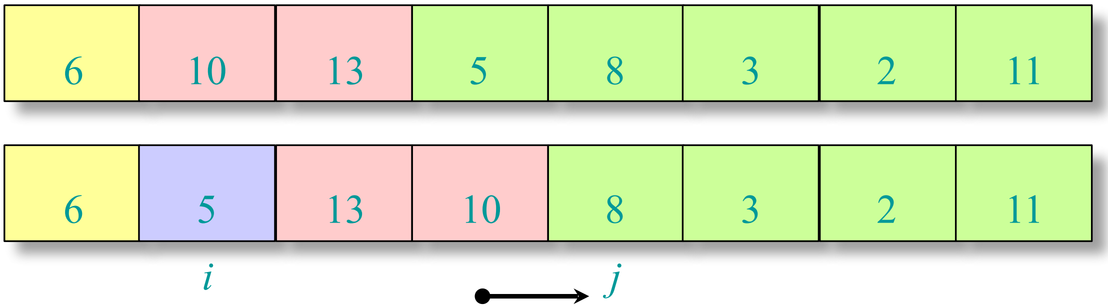
Example of partitioning



6	10	13	5	8	3	2	11

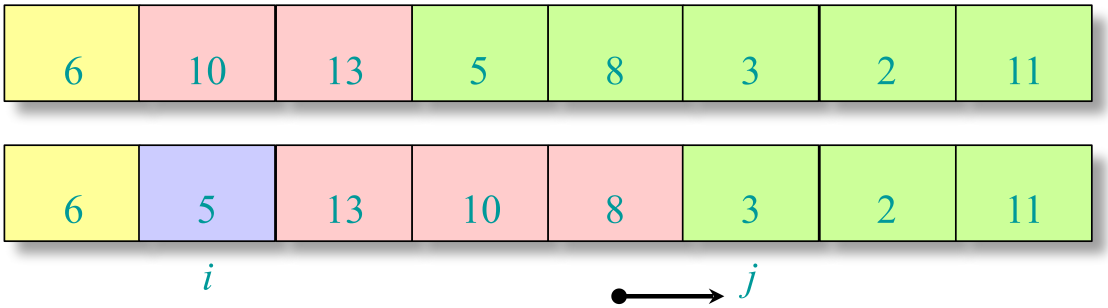
6	5	13	10	8	3	2	11

Example of partitioning



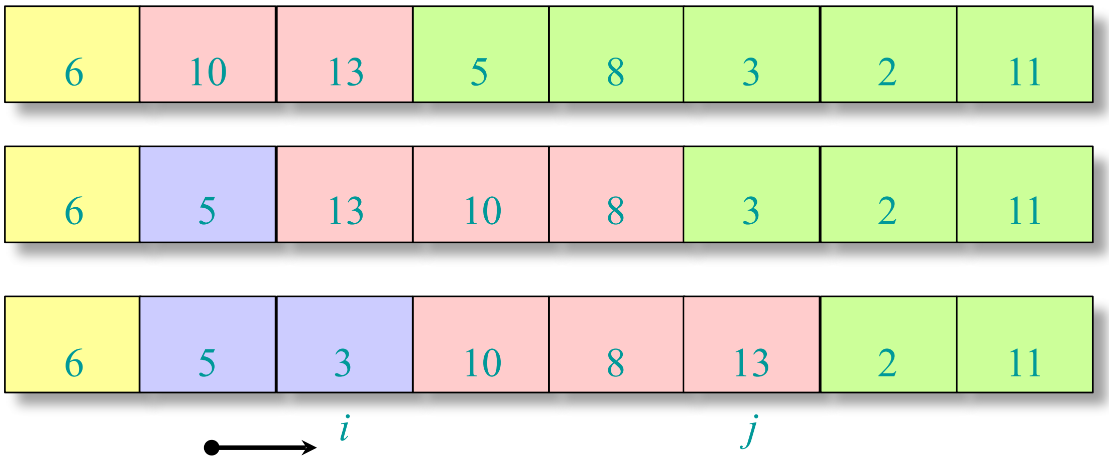
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11

Example of partitioning



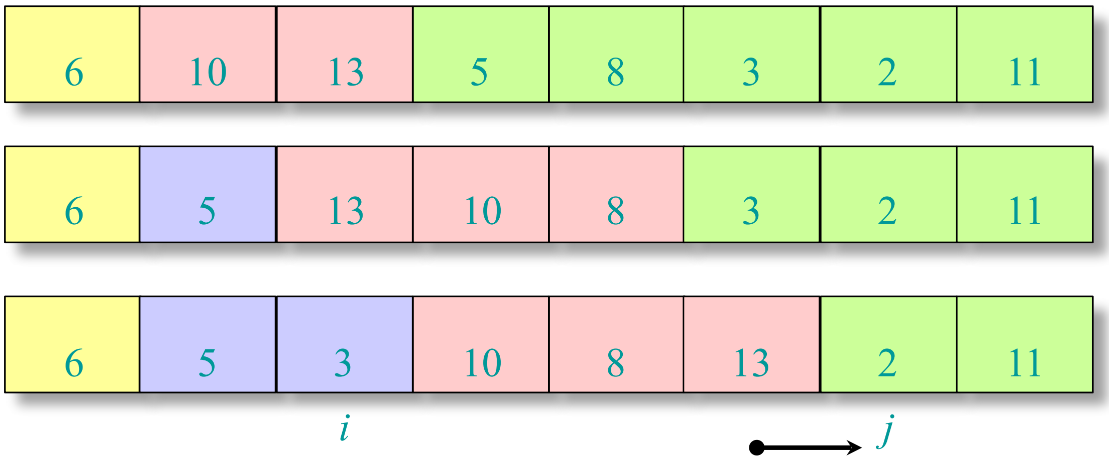
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
	i				*j*		

Example of partitioning



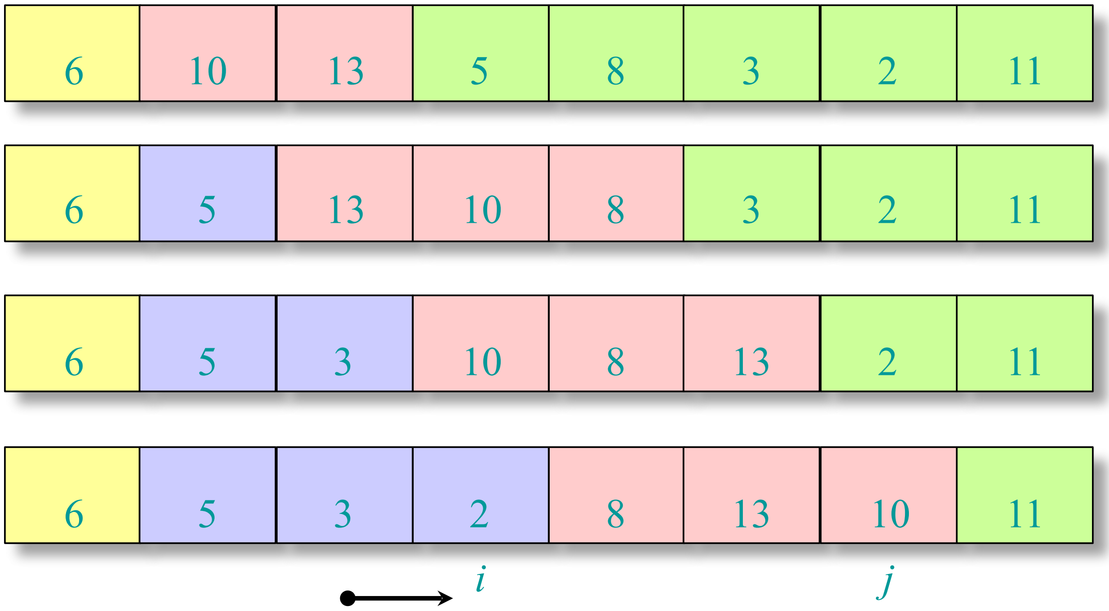
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
		i			*j*		

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
		i				*j*	

Example of partitioning



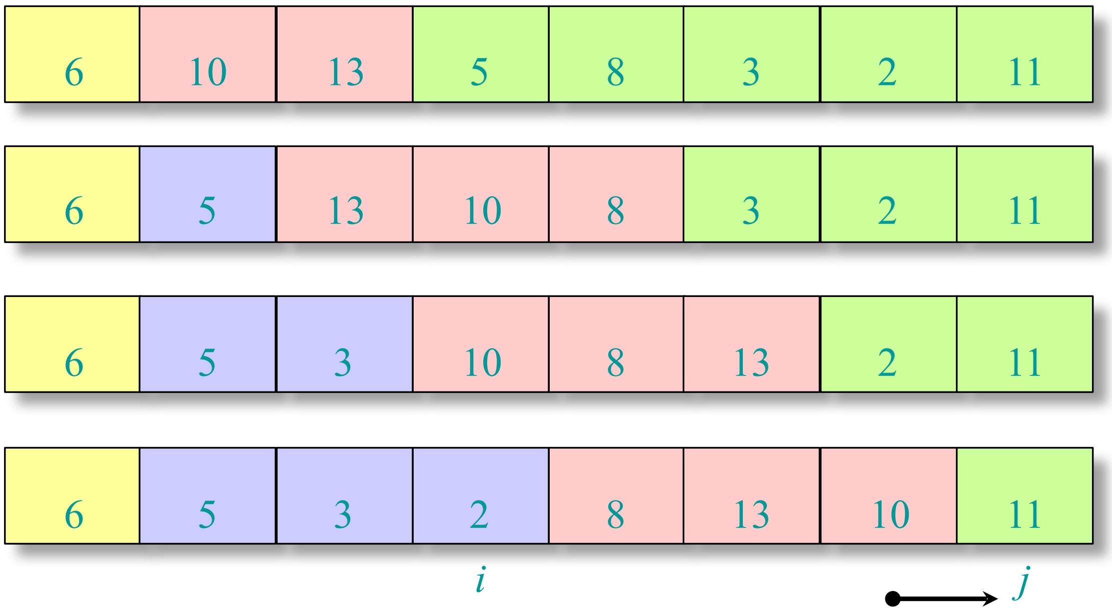
6 10 13 5 8 3 2 11

6 5 13 10 8 3 2 11

6 5 3 10 8 13 2 11

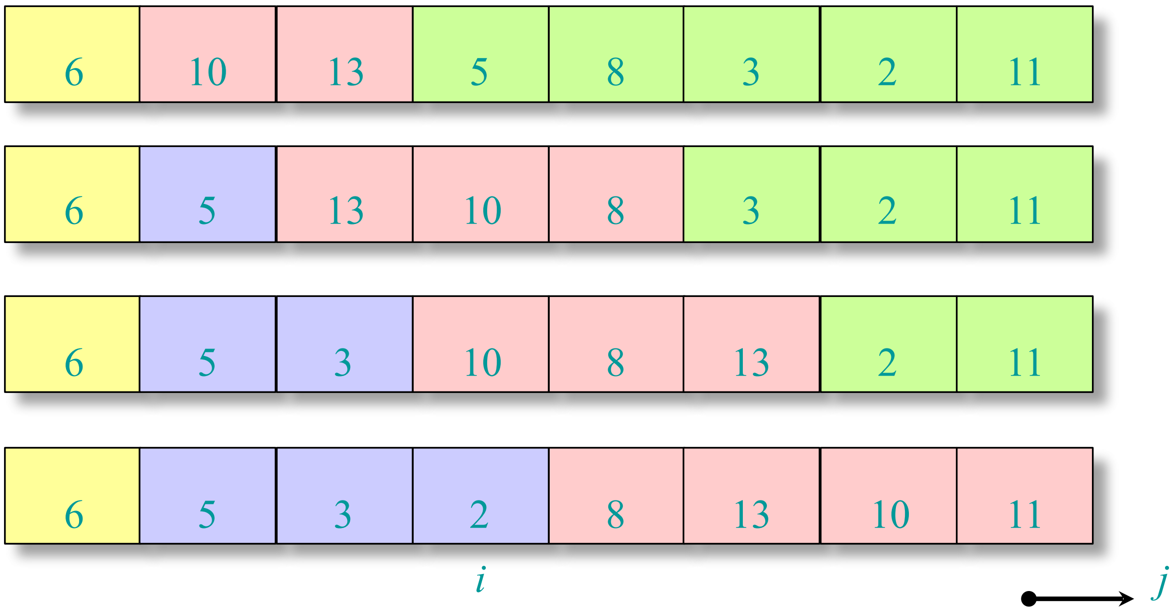
6 5 3 2 8 13 10 11

Example of partitioning



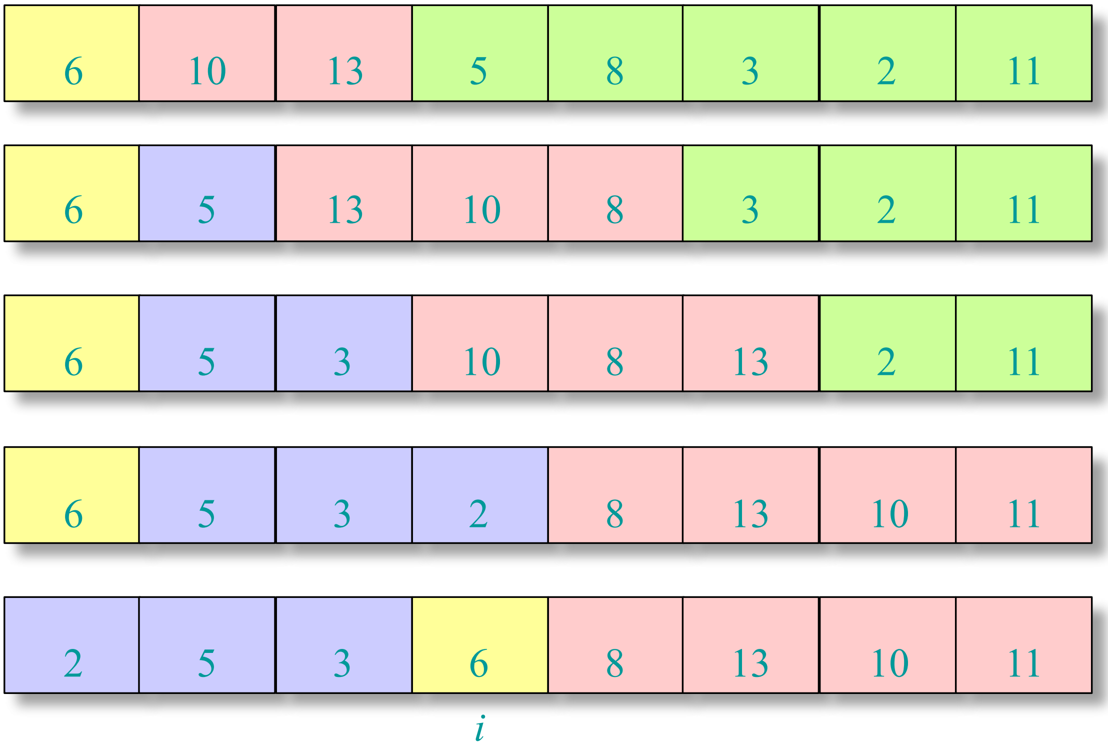
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

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

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 $p < r$ //do something only if contains at least 2 keys
        then $q \leftarrow$ PARTITION$(A, p, r)$  //both perform partition, and
            return index of pivot
            QUICKSORT$(A, p, q-1)$  //QS left part
            QUICKSORT$(A, q+1, r)$ //QS right part
```

Initial call: AUICKSORT$(A, 1, n)$

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

- 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

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

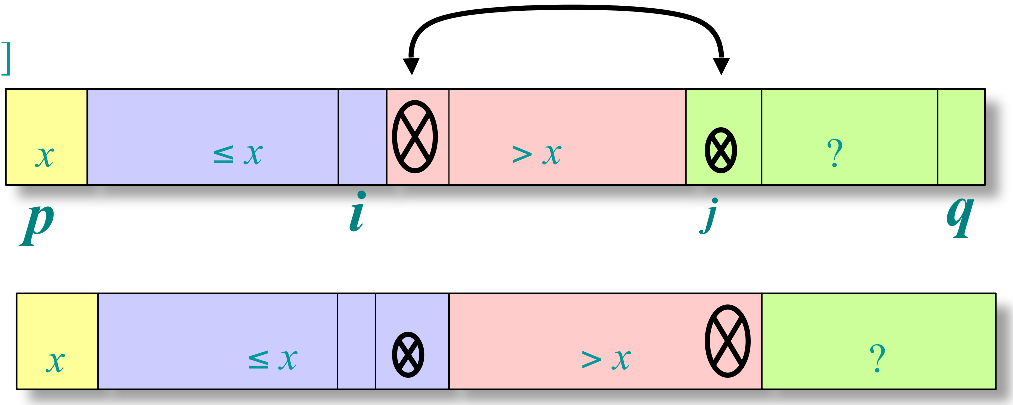
$$T(n) = n+(n-1)+(n-2)+(n-3)+,,,+1 \;=\; n(n+1)/2=\Theta(n^2)$$

Improving the Partitioning subroutine. Randomized partition.

```
PARTITION$(A, p, q)$              ▻ $A[p\,.\,.\,q]$
    $k$=rand(p,q) ▻  pick a random integer between $p$ and $q$.
    ▻  Careful-Dont pick a random value. Pick a random index
    exchange $A[i] \leftrightarrow A[k]$

    $x \leftarrow A[p]$                        ▻ pivot = $A[p]$
    $i \leftarrow p$
    for $j \leftarrow p+1$ to $q$                  ▻ j is hunting for small keys
        do if $A[j] \le x$                      ▻ Should send $A[j]$ to the left.
                then{
                     $i \leftarrow i+1$          ▻ Now $A[i]$>x
                     exchange $A[i] \leftrightarrow A[j]$ ▻ Fix $A[i]$>x
                }
    exchange $A[p] \leftrightarrow A[i]$
    return $i$
```

Invariant:



Best-case -the pivot is always the median (quite unlikely, but lets say)

If we are lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Lets understand where this bounds came from (amortized analysis)

We are paying a constant time to check a single key. Possibly another constant to move this key

Consider one of the keys.

The first time it is checked, it is in an array of size n

The second time it is checked, it is in an array of size n/2

The third time it is checked, it is in an array of size $n/4 = n/2^2$

..

The k'th time it is checked, it is in an array of size $\frac{n}{2^k}$ … note $k \le \log_2 n$

So it is checked (and possibly moved) $\log_2 n$ times.

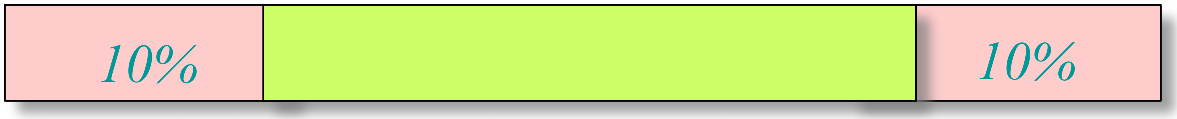
The total work for n keys is $O(n \log n)$

Randomized quicksort

How can find a pivot that guarantees partitions with good ratios for *A[1..n]*, ?

We say that *q* is a **good pivot** for if

- at least 10% of the elements of *A[1..n]* are smaller than *q,* and
- at least 10% of the elements of *A[1..n]* are larger than *q*.



What is the probability that when we pick a pivot, it is a good pivot? (Da ?)

So randomized QS finds good pivots on 80% of the time. The rest, it finds pivots that is less effective - we wasted only 20%.

Working with good pivots

What if the split is always no worse then $\frac{1}{10} : \frac{9}{10}$.

That is, the smaller of the two arrays contains at least *n/10* keys.

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

Consider one of the keys.

The first time it is checked, it is in an array of size ***n***

The second time it is checked, it is in an array of size $\leq 0.9n =$

The third time it is checked, it is in an array of size $\leq 0.9^2\ n$

..

The k'th time it is checked, it is in an array of size $\leq 0.9^k n$

How large could k be ? Solve

$n0.9^k \geq 1$ or $k = \log_{1.111} n = \log_2 n / \log_2 1.11 \leq 8 \log_2 n$

So it is checked (and possibly moved) O($\log_2 n$) times.

The total work for n keys is still $O(n \log n)$

Random Variable (light version)

- Assume we perform an experiment (Flipping a coin). Let **R** be the result – Face or Tail (F/T).
- We could define a random variable which (in this course) is a value that depends on the result of the experiment.
- Preferably, set to '1' if some condition is satisfied, and is `0' otherwise.
- Define **X** to be a random variable,
  set to 1 iff **R** is Face; **X**=0 if **R** is Tail.
- Define **Y** to be another random variable,
  - set to **Y** =5 iff **R** is Face; **Y**=-3 if **R** is Tail.

- We could ask what is the probability that **X=1**. Denote **Pr**(X=1)
- If coin is fair, **Pr(X=1)** is **0.5**, Pr(X=2)=Pr(X=3)=Pr(X=17)=0
- Pr(Y=1)=Pr(Y=2)=0 ; Pr(Y=5)=0.5 ; Pr(Y=-3)=0.5

Random Variable (light version)

- Assume we perform an experiment (tossing a dice). Let **R** be the result – one of the number 1,2,3,4,5,6.
- We could define a random variable which (in this course) is a value that depends on the result of the experiment.
- Preferably, set to ‘1’ if some condition is satisfied, and is `0’ otherwise.
- Define **F** to be a random variable, set to 1 iff R is even; (**F**=0 if cube falls on 1,3 or 5)
- Define **Q** to be another random variable, which is 1 iff R≥2.
- We could ask what is the probability that **F=1**. Denote **Pr**(F=1)
- If dice is fair, **Pr(F=1)** is **0.5**, and **Pr(Q=1)=5/6**

Random Variable and expectation (light version)

- In many cases, we would like to know what is the **expected** value of a random var.
- Example: If Y=1 we earn a dollar. What is the expected amount we earn in one game. We denote it by **E(Y)**
- $$E(Y)=\sum_{j=0}^{\infty} j\cdot Pr(Y=j)$$
- **Example:** Y is the value of the dice. We earn the value picked on the dice. So our expected earning is

$$\$1\cdot\frac{1}{6}+\ \$2\cdot\frac{1}{6}+\ \$3\cdot\frac{1}{6}+\ \$4\cdot\frac{1}{6}+\ \$5\cdot\frac{1}{6}+\$6\cdot\frac{1}{6}\ +\$7\cdot Pr(Y=7)+\ldots=\frac{\$21}{6}=\$3.5$$

- Good news. If **Y** is a Boolean var then **E(Y)**, the expected value of Y, is just **Pr(Y=1).**
- What if we earn $17 if Y=1.
- Lemma: for any constant ɑ it is always true that **E(ɑY) = ɑE(Y)**=ɑ**Pr(Y=1)**
- **Lemma** **E(X+Y+Z)=E(X)+E(Y)+E(Z)**

A warmup toy problem

```
$M \leftarrow -\infty$     $cnt \leftarrow 0$
for i=1…n {
    Pick a random value $r_i$ between (0,1)
        (informally and independently)
    If $M < r_i$ then {
        $M \leftarrow r_i$
        $cnt++$
        $x_i \leftarrow 1$
    }
    Else $x_i \leftarrow 0$ ;
}
```

```
$M \leftarrow 0$     $cnt \leftarrow 0$
Let A[1..n] be an array of keys, all different
Let B[1..n] be a random permutation of these keys. All
permutations are equally likely.
for i=1…n {
    If $M < B[i]$ then {
        $M \leftarrow B[i]$
        $cnt++$
        $x_i \leftarrow 1$
    }
    Else $x_i \leftarrow 0$ ;
}
```

Questions:

What is the minimum value of cnt?

What is the max value of cnt ?

But what we really want to know is what is the expected value of cnt

What is the expected value of cnt?

Note that always $cnt = \sum_{i=1}^{n} x_i$

$$E(x_i) = Pr(x_i = 1) = 1/i$$

$$E(cnt) = E(\sum x_i) = \sum E(x_i) = \sum_{i=1}^{n} 1/i = \ln n$$

An even better analysis for QS

The overall work of the algorithm is proportional to $\sum_{i=1}^{n}\sum_{j=1}^{i-1} y_{i,j}$

What we really want to know is what is the **expected** time, when we pick the pivot at random. $E\left(\sum_{i=1}^{n}\sum_{j=1}^{i-1} y_{i,j}\right).$

Luckily expectation of sum is sum of expectations:

$$E\left(\sum_{i=1}^{n}\sum_{j=1}^{i-1} y_{i,j}\right)=\sum_{i=1}^{n}\sum_{j=1}^{i-1} E\left(y_{i,j}\right) = \sum_{i=1}^{n}\sum_{j=1}^{i-1} Pr\left(y_{ij}=1\right)$$

Lemma: $Pr\left(y_{ij}=1\right) = \frac{2}{i-j+1}$ (**wake up** - this is the only point of the analysis where a new cool idea)

Proof - see next slide.

Using this lemma, we obtained

$$E\left(\sum_{i=1}^{n}\sum_{j=1}^{i-1} y_{i,j}\right) = \sum_{i=1}^{n}\sum_{j=1}^{i-1} \frac{2}{i-j+1} \leq 2n\left\{1+\frac{1}{2}+\frac{1}{3}+\ldots\frac{1}{n}\right\} = 2n\ln n.$$

Here we used the formula of the harmonic sum

Proving the Lemma

Let $S_{i,j} = \{j, j+1, j+2\ldots, i\}$. These are the keys between *j* and *i*.
Note that $|S_{i,j}| = i - j + 1$.
Consider the first pivot *x* that we pick from $S_{i,j}$. (other pivots are not relevant to $Pr(y_{ij} = 1)$). There are three cases:

- **Case 1**: this pivot *x* is *i*. Then *x* will be compared to all keys of $S_{i,j}$, including *j*. $y_{ij} = 1$ in this case.
- **Case 2**: *x* is *j*. Then it will be compared to all keys of $S_{i,j}$, including *i*. $y_{ij} = 1$ in this case.
- **Case 3**: *x* is one of the other keys in $S_{i,j}$. (that is, $j < x < i$). In this case, the partition separates *i* from *j*. In the future calls to 'partition' i and j are in different subarrays. They will not be compared, and $y_{ij} = 0$.

Since each keys has the same probability to be picked as a pivot,

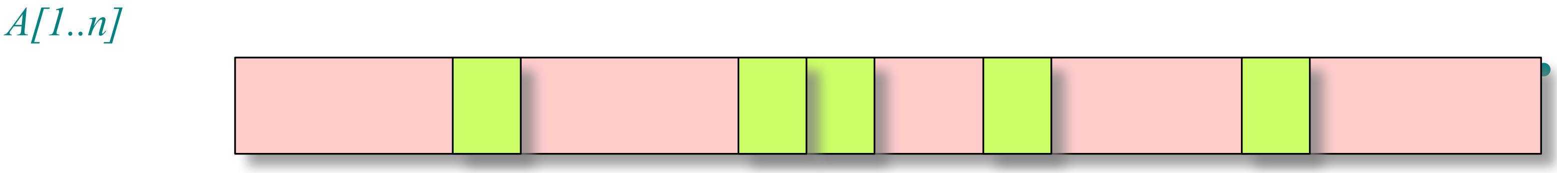
$$Pr(y_{i,j} = 1) = \frac{2}{|S_{ij}|} = \frac{2}{i - j + 1}. \qquad \text{QED}$$

Finding a good pivot for *A[1..n]*

5-random-elements method. :

- Pick the **indices** of $\leq$ 5 elements at random from *A[1..n],*
- *For k=1 to 5*

$$X[k] = A\left[\lfloor n \cdot \ rand() \rfloor\right]$$



- Set q to be the median of *X[1..5]*

Quicksort in practice

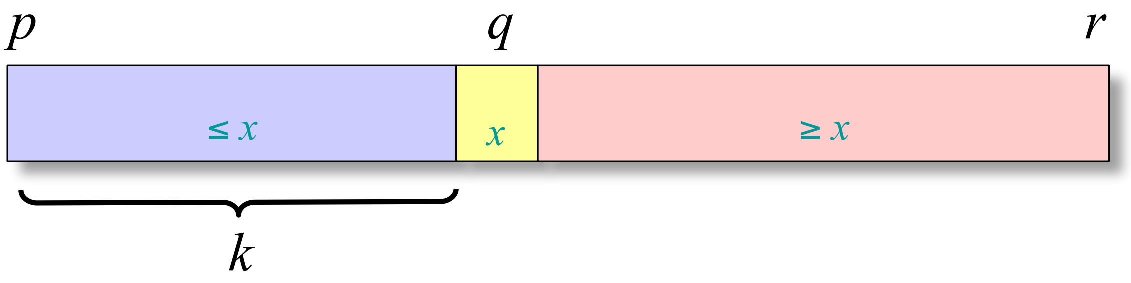
- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort behaves well even with caching and virtual memory.

Median Selection

- (CLRS Section 9.2, page 185).
- For *A[1..n]* (all different elements) we say that the rank of *x* is ***i*** if exactly ***i-1*** elements in *A* are smaller than *x*.
- In particular, the median is the $\lfloor n/2 \rfloor$-smallest.
- To find the median, we could sort and pick $A[\lfloor n/2 \rfloor]$ (taken O(*n* log *n*)).
- We can do better.

Median Selection-cont

```
RS( $A, p, r, i$){
    //Randomize Selection: Returns $i$'st smallest element in $A[p..r]$.
    //Assumption: Input is valid and elements are different.
•If $p==r$ return A[$p$]
•$q$=PARTITION($A,p,r$) ;
    •//Partition using the 5-random element method
•$k=q-p$
•If $i==k+1$ return $A[q]$
•If $i<k$ return RS($A, p, q-1, i$ ) // Note the difference from QS
•Else return RS($A, q+1, r, i-k-1$)
}
```



Time analysis

- Recall: With high probability, we pick a good pivot:
  - Not in the 10% smallest or largest:
- Hence, we get rid of at least 10% of the elements of A
- So, $T(n)=cn+T(0.9\ n)$.
  - $T(n)=c(n+0.9n+ 0.9^2n+0.9^3n+\ldots) =$ $cn(1+0.9+ 0.9^2+0.9^3+\ldots) =$ $cn(1/(1-0.9)) = O(n)$.
- So the expected time is linear. (yuppie)

As in the case of QS, partitions which are not good are not harmful, just not helpful.