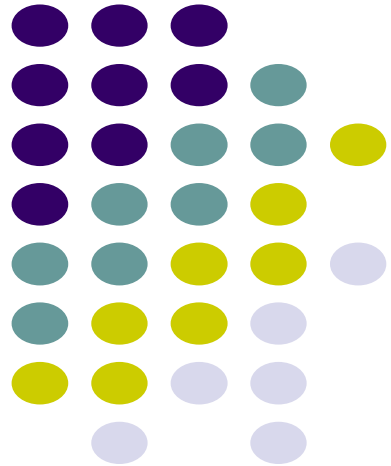
Randomized Algorithms

Dr. Navjot Singh

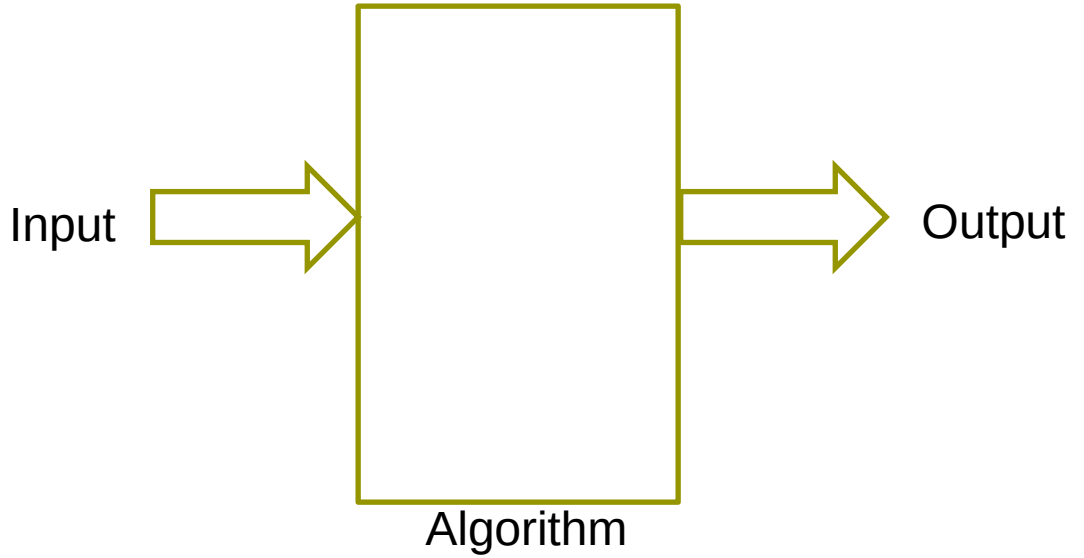
Design and Analysis of Algorithms



Acknowledgements

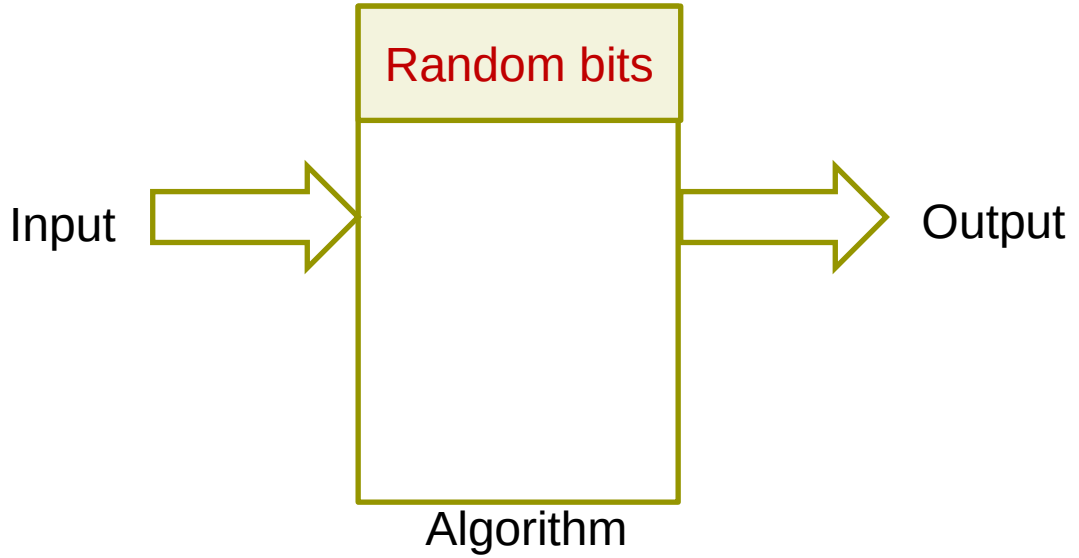
- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill

Deterministic Algorithm



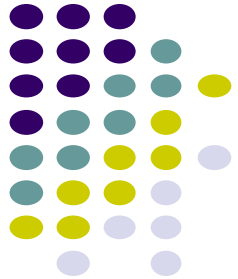
- The **output** as well as the **running time** are **functions** only of the **input**.

Randomized Algorithm



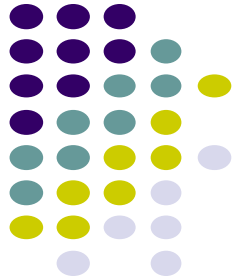
- The **output** or the **running time** are **functions** of the **input** and **random bits chosen**.

Why use randomness?



- Avoid worst-case behavior: randomness can (probabilistically) guarantee average case behavior
- Efficient approximate solutions to intractable problems

Types of Randomized Algorithms



Randomized Las Vegas Algorithms:

- Output is always correct
- Running time is a **random variable**

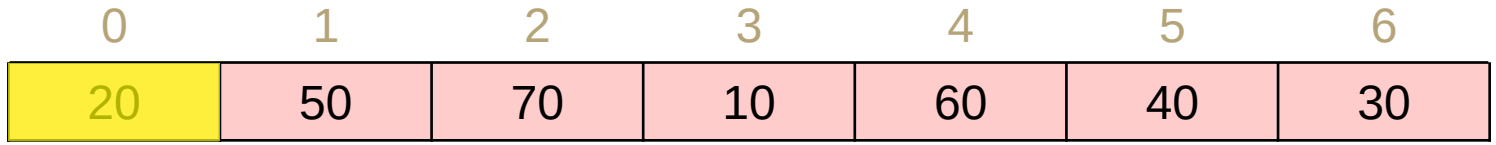
Example: Randomized Quick Sort

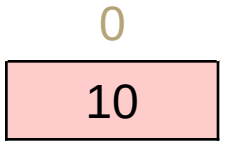
Randomized Monte Carlo Algorithms:

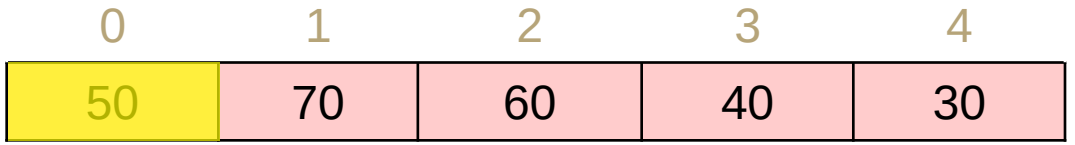
- Output may be incorrect with some probability
- Running time is deterministic.

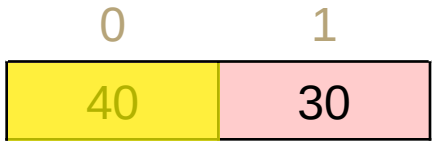
Example: Randomized algorithm for approximate median

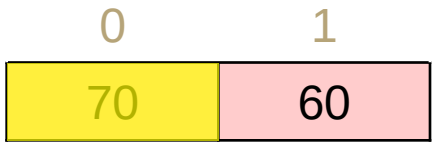
Quick Sort

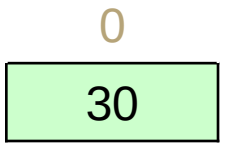

0 1 2 3 4 5 6
20 50 70 10 60 40 30

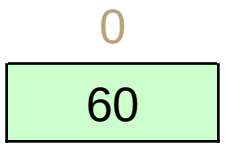

0
10

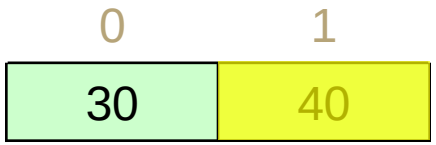

0 1 2 3 4
50 70 60 40 30


0 1
40 30

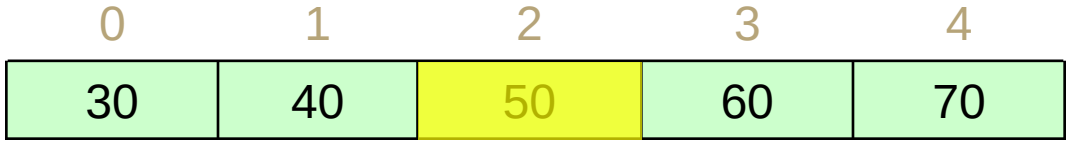

0 1
70 60

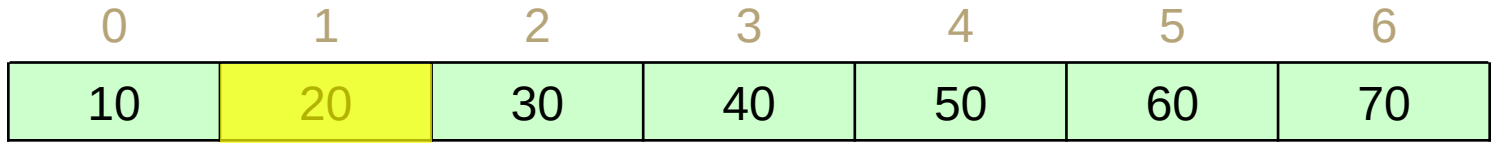

0
30


0
60


0 1
30 40


0 1
60 70


0 1 2 3 4
30 40 50 60 70

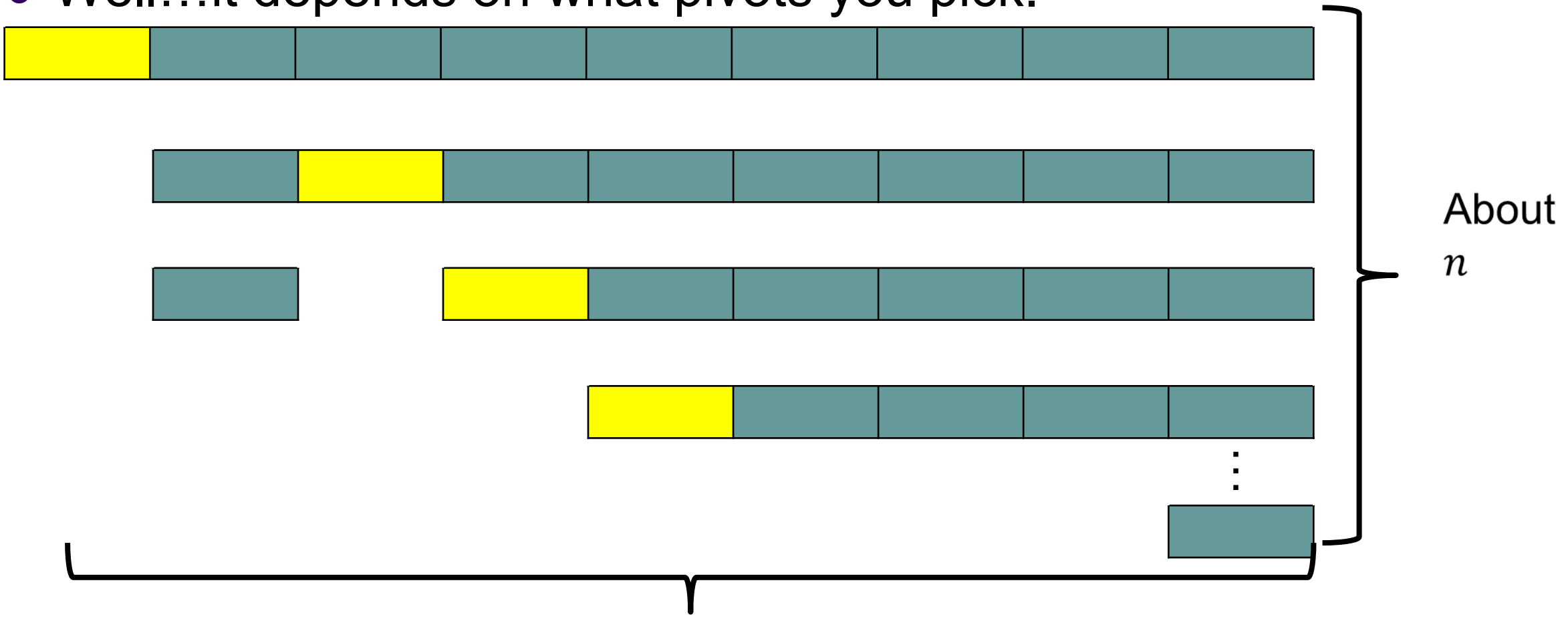

0 1 2 3 4 5 6
10 20 30 40 50 60 70

How long does it take?

Total time:

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

- Well…it depends on what pivots you pick.



QuickSort(S)

- QuickSort(S)
{ If (|S|>1)
Pick and remove an element x from S;
($S_{<x}$, $S_{>x}$)← Partition(S,x);
return(Concatenate(QuickSort($S_{<x}$), x, QuickSort($S_{>x}$))
}

QuickSort(S)

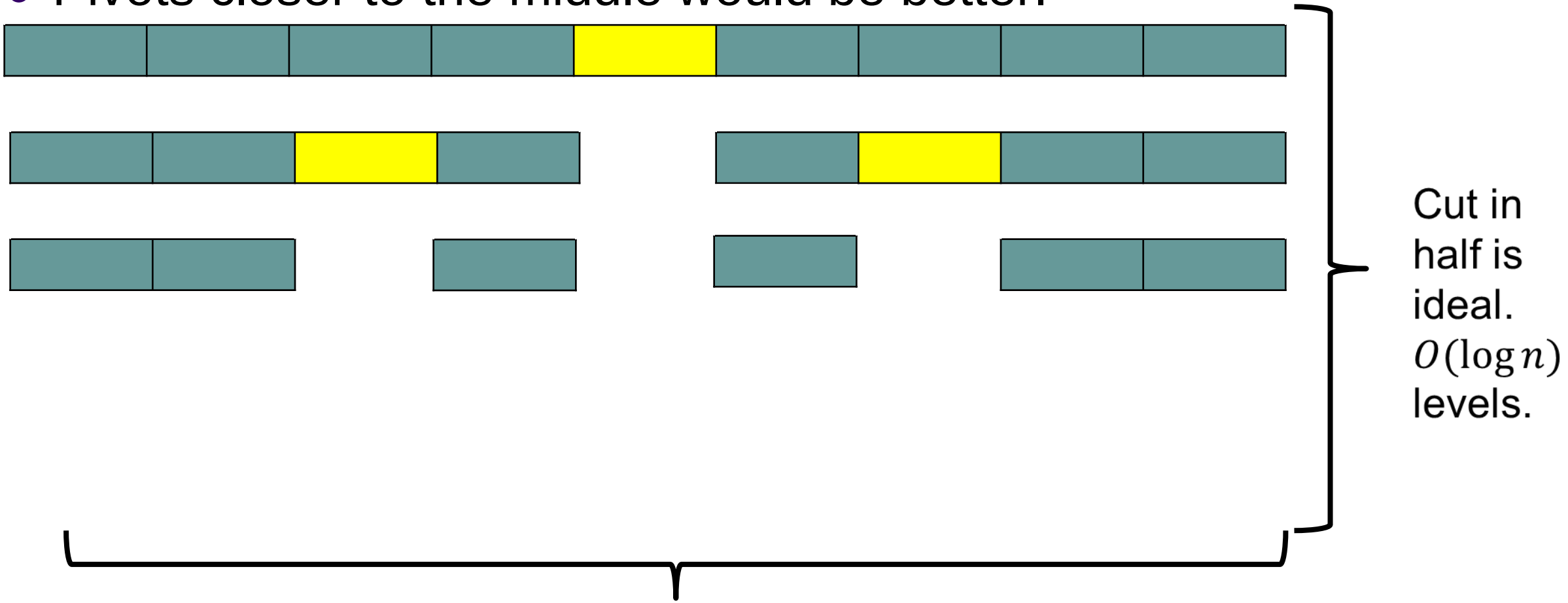
When the input S is stored in an array A

```
QuickSort(A, l, r)
{       If (l < r)
                x ← A[l];
                i ← Partition(A, l, r, x);
                   QuickSort(A, l, i − 1);
                   QuickSort(A, i + 1, r)
}
```

- **Average** case running time: **O(n log n)**
- **Worst** case running time: **O(n^2)**
- **Distribution sensitive**: Time taken depends upon the initial permutation of A.

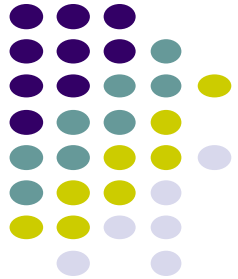
What leads to a good time?

- Pivots closer to the middle would be better.



Randomized QuickSort(S)

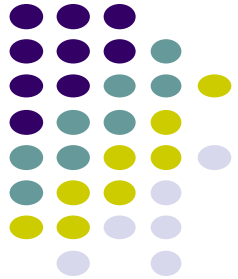
When the input S is stored in an array A



```
QuickSort(A, l, r)
{       If (l < r)
            x ← an element selected randomly uniformly from A[l..r];
            i ← Partition(A, l, r, x);
            QuickSort(A, l, i − 1);
            QuickSort(A, i + 1, r)
}
```

- **Distribution** insensitive: Time taken does **not** depend on initial permutation of A.
- Time taken **depends** upon the **random** choices of pivot elements.
  1. For a given input, Expected(**average**) running time: **O(n log n)**
  2. **Worst** case running time: **O(n^2)**

Common Quicksort Implementations



- A common strategy in practice is the “median of three” rule.
- Choose three elements (either at random or from specific spots). Take the median of those for your pivot
- Guarantees you don’t have the worst possible pivot.
- Only a small constant number of extra steps beyond the fixed pivot (find the median of three numbers is just a few comparisons).