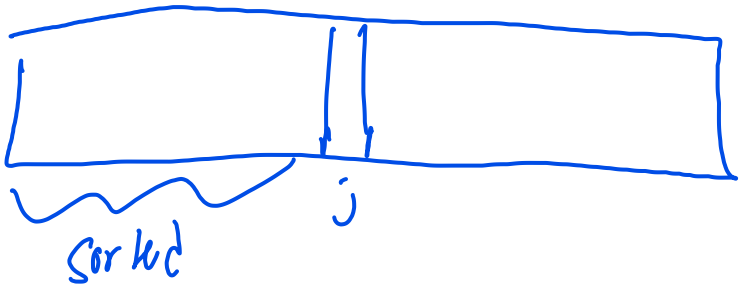
Last time we considered the average case running time of a deterministic algorithm.

That is, we assumed that the input was uniformly distributed (any permutation of the input is equally likely to occur in the case of the hiring problem), and we analyzed the running time of the algorithm of a randomly chosen input.

Consider the average case of insertion sort when any permutation of the numbers in the array is equally likely to occur.



$\frac{j}{2}$ swaps on average

$$\sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2} \sum_{j=2}^{n} j = \Theta(n^2)$$

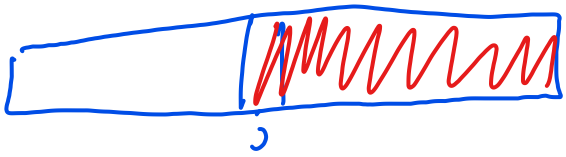
There are a few drawbacks to analyzing algorithms in this way.

- The distribution of inputs are heavily reliant on the application. Therefore the analysis may be misleading for certain applications.
- For many applications, it is difficult to determine if the inputs come from some "well-behaved" distribution, and if we are wrong about the distribution then our analysis of the running time could be significantly off base.

One approach to deal with this would be to "shuffle" the input before running the algorithm. Essentially we can force the input to come from a known distribution. Of course the downside to this is we pay a price in the running time of the algorithm.

A better approach to deal with this is to use a **randomized algorithm**. That is, an algorithm which uses a random number generator to determine some of the steps of the algorithm.

- For example in insertion sort, instead of inserting the element in position j, randomly choose a one of the last $n - j$ elements to insert.



The running time of a randomized algorithm is a random variable. We are interested in determining the **expected running time** of the algorithm. That is, let X be the random variable that measures the running time of a randomized algorithm. We are interested in bounding $E[X]$.

Now we do not need to make any assumptions about the input distribution.

No specific input will force the algorithm's worst-case behavior (although no input will force the algorithm's best-case behavior either). The worst-case (and best-case) are determined only by the output of a random number generator.

For these reasons, it is generally considered better to analyze the expected running time of a randomized algorithm rather than the average case running time of a deterministic algorithm.

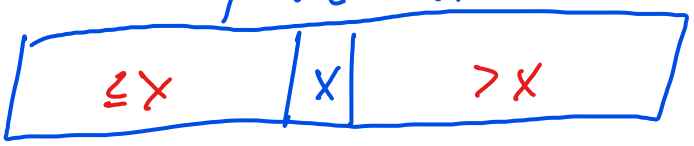
Quicksort: A divide-and-conquer sorting algorithm (like mergesort).

Quicksort is an "in place" algorithm (limited extra space is needed, unlike mergesort).

We are going to first analyze the deterministic quicksort algorithm, and then perform an expected running time analysis on the randomized version of quicksort.

Quicksort

1) Divide: Partition array into two subarrays around a pivot x such that the elements in first subarray are $\leq x$ and elements in second subarray are $> x$.

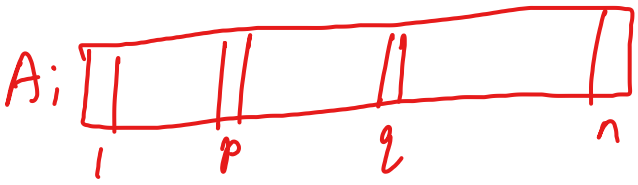


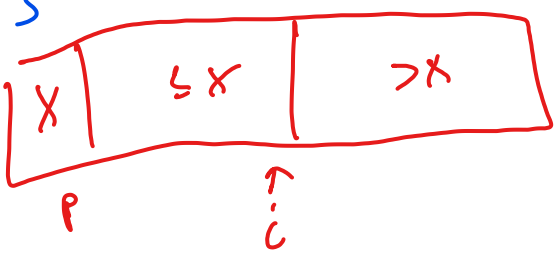
$\Theta(n)$ time.

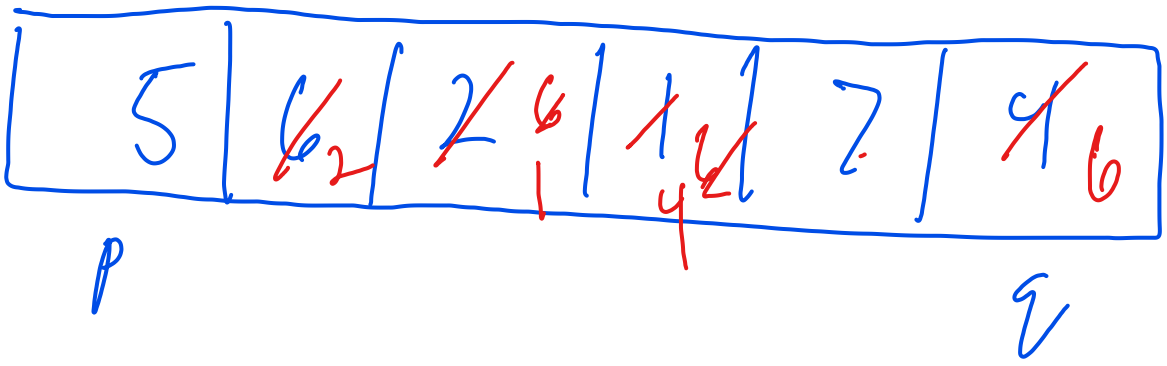
2) Conquer: Recursively sort subarrays.

3) Combine: Trivial.

```
Partition (A, p, q) {
    x = A[p]
    i = p
    for (j = p+1 to q)
    {
        if (A[j] ≤ x)
        {
            i++
            swap(A[i], A[j])
        }
    }
    swap(A[p], A[i])
    return i
}
```

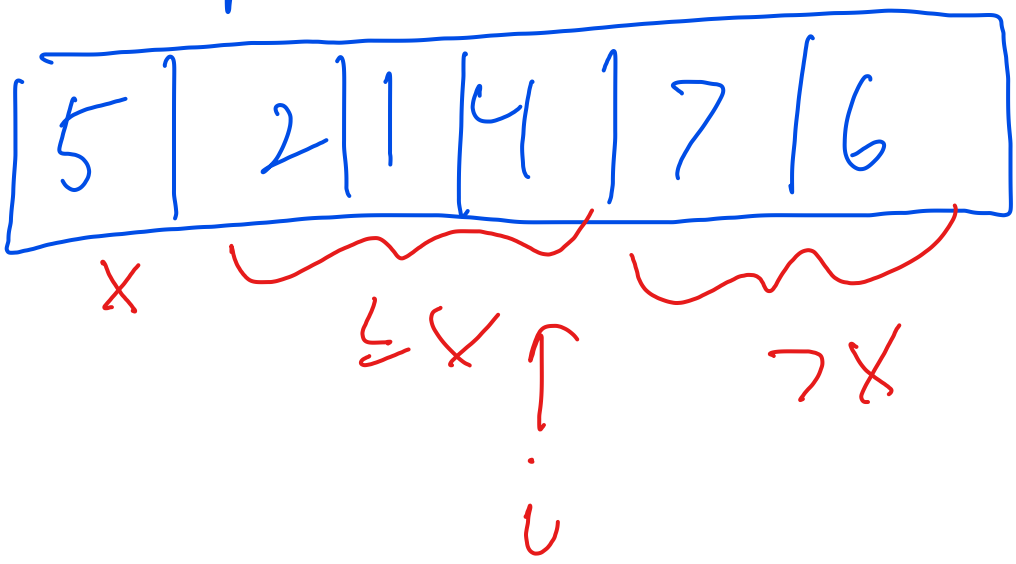




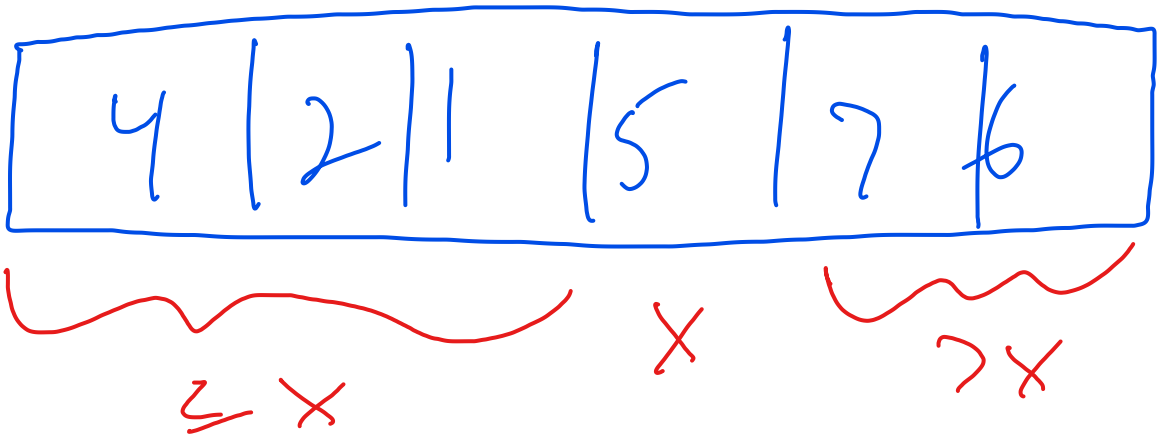


x i j
5 p p+1
p+1 p+2
p+2 p+3
p+3 p+4
p+5

After loop:



At end:

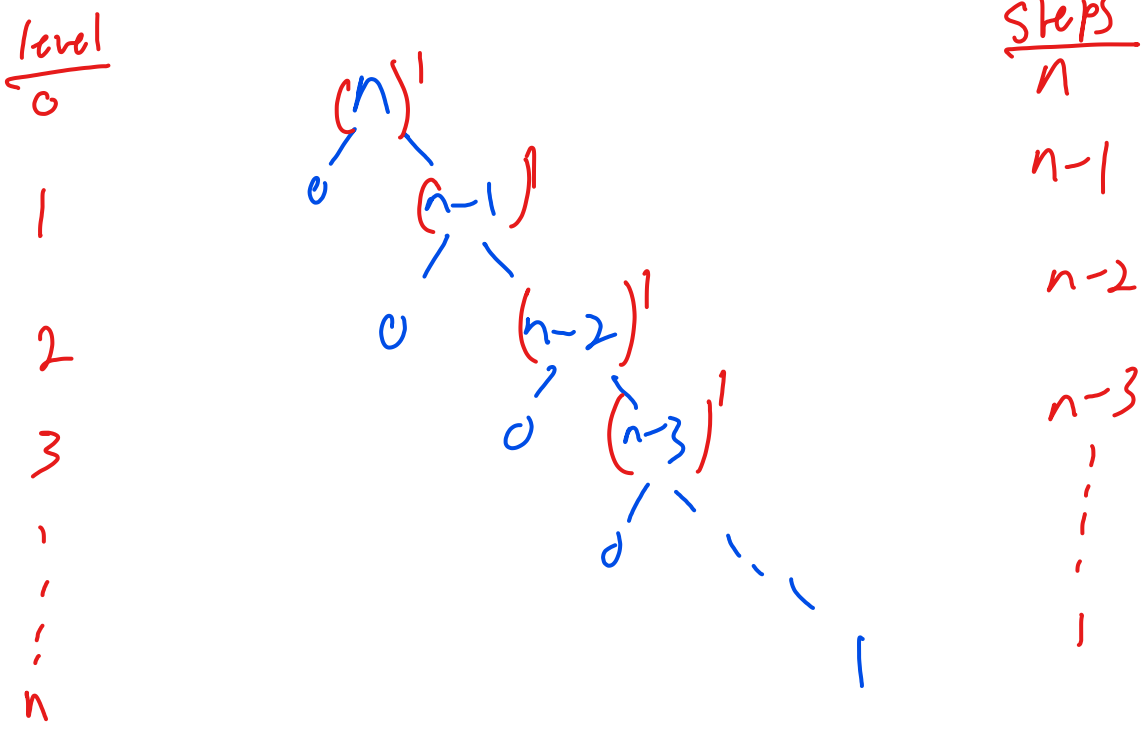


```
QuickSort(A, p, r)
{
    if(p < r)
    {
        q = Partition(A, p, r)
        (QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)
    }
}
```

Call QuickSort(A, 1, n) from main().

Worst Case: Pivot is smallest or largest # overall.

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



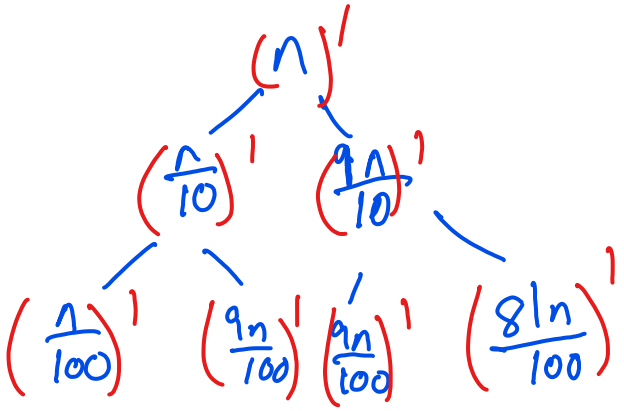
$$\sum_{i=0}^{n} i = \Theta(n^2)$$

Best Case: pivot is median every time

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow \Theta(n \log n)$$

---

what if we get $\frac{n}{10}$ and $\frac{9n}{10}$ splits?



Steps

n

$\frac{n}{10} + \frac{9n}{10} = n$

$\frac{n}{100} + \frac{9n}{100} + \frac{9n}{100} + \frac{81n}{100} = n$

$\vdots$

n

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

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