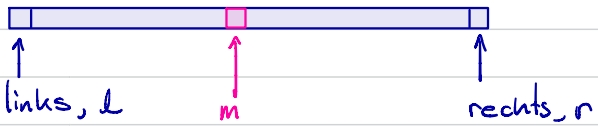
Binary Search

→ effiziente Suche (vs. ineffizient z.B. lineare Suche)

```
def bin_search(a, l, r, b):
  if r < l:                          } l muss immer links von r sein
    return None
  else:
    m = (l + r) // 2 → neue Mitte
    if a[m] == b:                    } Element gefunden
      return m
    elif b < a[m]:                   } gesuchtes Elem. kleiner als a[m] → in linker Hälfte weiter suchen
      return bin_search(a, l, m-1, b)
    else: # a[m] > b
      return bin_search(a, m+1, r, b)   → rechte Hälfte
```



Quicksort

→ kann iterativ und rekursiv Implementiert werden
(siehe e.g. Thread hier https://stackoverflow.com/questions/12553238/quicksort-iterative-or-recursive)
↳ meistens wird die rekursive Implementation betrachtet.

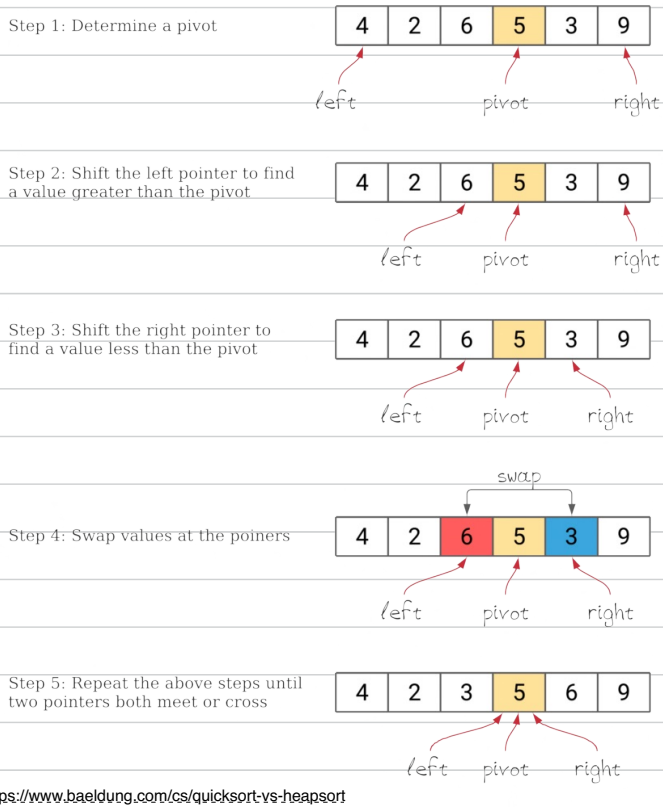
→ Schritte:

1) Pivot p Element wählen → hat Einfluss auf Effizienz
   ↳ erstes Elem, letztes Elem
   ↳ zufälliges Elem, Mitte
   ↳ Median einer gewissen Anzahl Elem
   e.g. median(a[l], a[r], a[(l+r)//2])

2) Liste partitionieren (partition algo.)
   ↳ links von p sind alle Elemente kleiner als p, rechts alle grösser

3) Schritte 1) und 2) auf den Sublists wiederholen.

→ Bsp / Illustration Partition:



https://www.baeldung.com/cs/quicksort-vs-heapsort

https://mycareerwise.com/programming/category/sorting/quick-sort-using-recursion

(aus den Slides)

```
def quicksort(a, l, r):
    if l < r:
        k = partition(a, l, r)
        quicksort(a, l, k - 1)
        quicksort(a, k + 1, r)
```

```
def partition(a, l, r):
  p = a[r]
  j = l
  for i in range(l, r):
    if a[i] < p:
      a[i], a[j] = a[j], a[i]
      j += 1
  a[j], a[r] = a[r], a[j]
  return j
```

Invariante: Vor Iteration i ...

- Alle Elemente in `a[l:j]` sind $< p$.
- Alle Elemente in `a[j:i]` sind $\geq p$.

Laufzeit: $\Theta(r-l) \subseteq \Theta(n)$.