Algorithm analysis & design

Analysis Of Recursive Algorithms

Presented By:

T.A. **Asmaa Hamad El-saied**

E-mail: **eng.asmaa134@gmail.com**

Agenda

- **Analysis of recursive algorithms**
  - Iteration (Substitution) method
  - Master Method (Theorem)
  - Recursion tree method
- **Examples of Recursive Algorithms**
- **Divide and Conquer**
  - Merge Sort
  - Quick Sort

General form of recursive function

f(N)

{

Recursive: f( ), f( ),…..

}

Non recursive part

How to Analyze the Recursive Functions?

- **Two Steps**

1. Compute running time (T(N)) (**recurrence equation**) the recursive function

   T(n)=T(non recursive part)+ T(different N)

2. Solve T(n) recurrence relation using one of Two methods to get the order (**Θ** or **O**)

   1. **Iteration Method**
   2. **Master method**

First Step

Examples

- **Compute the recurrence of the following examples (just deduce T(N)):**
  - Factorial
  - Fibonacci
  - Binary search

Examples

- **Factorial**

```
ALGORITHM Fact (N)
if N= =1 or N==0
        return 1
else
        return Fact(N-1) *N
```

- **T(N)= C+T(N-1)**

Examples

- **Fibonacci**

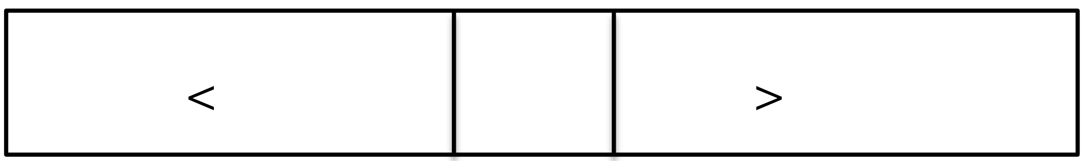
```
fibonacci (N)
 {
   if N== 0
       return 0
    Else if N==1
      return 1
    else
    return fibonacci(N – 1) + fibonacci(N – 2);
 }
```

- **$T(N)= C+T(N-1)+T(N-2)$**

Examples

- **Binary search**

item



Sorted

- **T(N)= C+T(N/2)**

Second step

Iteration (Substitution)

Iteration Method

- Idea…
  - Deduce the complexity by iteratively substitute the T(N) terms until reaching general formula. Then, get the complexity using both the formula and the termination condition.

Iteration Method

- When to Use?!
  - Deduction of general formula is easier if there's exist **one term** in the recurrence. (i.e. recurrence is in the form of

    T(N) = a **T(something)**+ something2

    (One T: T(something))
  - and **NOT** appropriate if

    T(N)= a **T(something1) + b T(something2) + … +** something

    - (More than One T: T(something1),T(something2) …)
    - (It will be **difficult** to deduce its general formula)

Example

- **Solve this recurrence**

  T(N) = T(√N) + c ;T(2) = 0

- **Solution**

$$T(N) = T(N^{0.5}) + c \qquad ;T(2) = 0 \qquad \rightarrow (\mathrm{I})$$

$$T(N^{0.5}) = T\left(N^{0.5^2}\right) + c \qquad ;\text{substitute in } (\mathrm{I})$$

$$T(N) = T\left(N^{0.5^2}\right) + c + c = T\left(N^{0.5^2}\right) + 2c \qquad \rightarrow (2)$$

$$T\left(N^{0.5^2}\right) = T\left(N^{0.5^3}\right) + c \qquad ;\text{substitute in } (2)$$

$$T(N) = T\left(N^{0.5^3}\right) + c + 2c = T\left(N^{0.5^3}\right) + 3c \qquad \rightarrow (3)$$

$$T\left(N^{0.5^3}\right) = T\left(N^{0.5^4}\right) + c \qquad ;\text{substitute in } (3)$$

$$T(N) = T\left(N^{0.5^4}\right) + c + 3c = T\left(N^{0.5^4}\right) + 4c \qquad \rightarrow (4)$$

Example

Generally at K:

$$T(N) = T\left(N^{0.5^K}\right) + K \times c$$

<u>Termination:</u>

$$T\left(N^{0.5^K}\right) = T(2) \rightarrow N^{0.5^K} = 2 \qquad ;\text{take } \log_2 \quad \text{for both sides}$$

$$\log_2 N^{0.5^K} = \log_2 2 \rightarrow 0.5^K \times \log_2 N = 1$$

$$\rightarrow 0.5^K = \frac{1}{\log_2 N} \rightarrow \left(\frac{1}{2}\right)^K = \frac{1}{\log_2 N}$$

$$\rightarrow (2)^K = \log_2 N \qquad ;\text{take } \log_2 \quad \text{for both side}$$

$$\rightarrow K = \log_2(\log_2 N)$$

Substitute in general formula:

$$T(N) = T(2) + \log_2(\log_2 N) \times c$$

$$T(N) = 0 + \log_2(\log_2 N) \times c$$

$$\boldsymbol{T(N) = \theta(\log(\log N))}$$

MASTER THEOREM

Master method

- Idea…
  - Given a recurrence of the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
  - where, $a \geq 1$, $b > 1$, and $f(n) > 0$
  - Compare $f(n)$ vs $n^{\log_b a}$
    - ✓ **Case 1:** if $f(n) = O(n^{\log_b a - \varepsilon})$;$\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$
    - ✓ **Case 2:** if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$
    - ✓ **Case 3:** if $f(n) = \Omega(n^{\log_b a + \varepsilon})$;$\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then: $T(n) = \Theta(f(n))$

Master method

- When to Use?!

  If the recurrence is in the form of

  $T(N)= aT(N/b)+f(N)$

Examples

- **EX1: $T(N) = 7\,T(2N/3) + N^2 \log(N)$**

Solution

a = 7, b = 3/2, f(N) = N^2 log(N)

Compare f(N) vs. $N^{\log_b a}$

$$N^{\log_b a} = N^{\log_{3/2} 7} = N^{2.8} \quad \text{vs.} \quad f(N) = N^2 \log N$$

$N^{\log_b a} >$ f(N) ➔ It's case (1), then we **must** find $\varepsilon > 0$

Examples

Case (1): *f(n) is* $O(n^{\log_b a - \varepsilon})$

$$N^2 \log N = O(N^{2.8-\varepsilon}) = O(N^2 \times N^{0.8-\varepsilon})$$

$$\rightarrow \log N = O(N^{0.8-\varepsilon})$$

But we have: $(\log N)^a$ *is* $\boldsymbol{o}(N^b)$ for any constant *a, b* > 0.

$$\rightarrow any\ 0 < \varepsilon < 0.8\ will\ satisfy\ \log N = O(N^{0.8-\varepsilon})$$

- Master method **Succeeded**
- $\boldsymbol{T(N) = \theta\left(N^{2.8}\right)}$

Examples

- **EX2:** $T(n) = 2T(n/2) + n$

  $a = 2,\ b = 2,\ \log_2 2 = 1$

  Compare $n^{\log_2 2}$ with $f(n) = n$

$N^{\log_b a} = f(N)$ ➔ It's case (2),

Case 2

$\Rightarrow f(n) = \Theta(n) \Rightarrow T(n) = \Theta(n\lg n)$

Examples

- **EX3: $T(n) = 2T(n/2) + n^2$**

  $a = 2$, $b = 2$, $\log_2 2 = 1$

  Compare n with $f(n) = n^2$

$N^{\log_b a} < f(N)$ ➔ It's case (3), then we **must** find $\varepsilon > 0$

Case 3

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon}) \rightarrow$ $any\ 0 < \varepsilon < 1\ will\ satisfy$ $n^2 = \Omega(n^{1+\varepsilon})$

$\Rightarrow a\ f(n/b) \leq c\ f(n)$

$\Leftrightarrow 2\ n^2/4 \leq c\ n^2 \Rightarrow c = ½$ is a solution ($c<1$)

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

Examples

- **EX4: $T(n) = 2T(n/2) + n^{1/2}$**

$a = 2$, $b = 2$, $\log_2 2 = 1$

Compare n with $f(n) = n^{1/2}$

$N^{\log_b a} > f(N)$ ➔ It's case (1), then we **must** find $\varepsilon > 0$

Case 1

$\Rightarrow f(n) = O(n^{1-\varepsilon}) \rightarrow$ $any\ 0 < \varepsilon < 0.5\ will\ satisfy$ $n^{1/2} = O(n^{1-\varepsilon})$

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

Examples

- **EX5: T(n) = 3T(n/4) + nlgn**

  $a = 3$, $b = 4$, $\log_4 3 = 0.793$

  Compare $n^{0.793}$ with $f(n) = nlgn$

  $N^{\log_b a} < f(N)$ ➔ It's case (3), then we **must** find $\varepsilon > 0$

Case 3

$\Rightarrow f(n) = \Omega(n^{0.793+\varepsilon}) \rightarrow$ $any\ 0 < \varepsilon < 0.2\ will\ satisfy$ $nlgn = \Omega(n^{0.793+\varepsilon})$

$\Rightarrow a\, f(n/b) \leq c\, f(n)$

$3*(n/4)lg(n/4) \leq c *nlgn$ ➔ $(3/4)nlgn \leq c *nlgn$, $c=3/4 < 1$

$\Rightarrow T(n) = \Theta(nlgn)$

Divide and Conquer

Divide and Conquer

- Divide-and conquer is a general algorithm design paradigm/model;
  - **Divide:** divide the problem into small sub-problem(s)
  - **Conquer :** Solve the sub-problems recursively in same manner
  - **Combine :** the solution of the sub-problems to get the final solution

Merge Sort

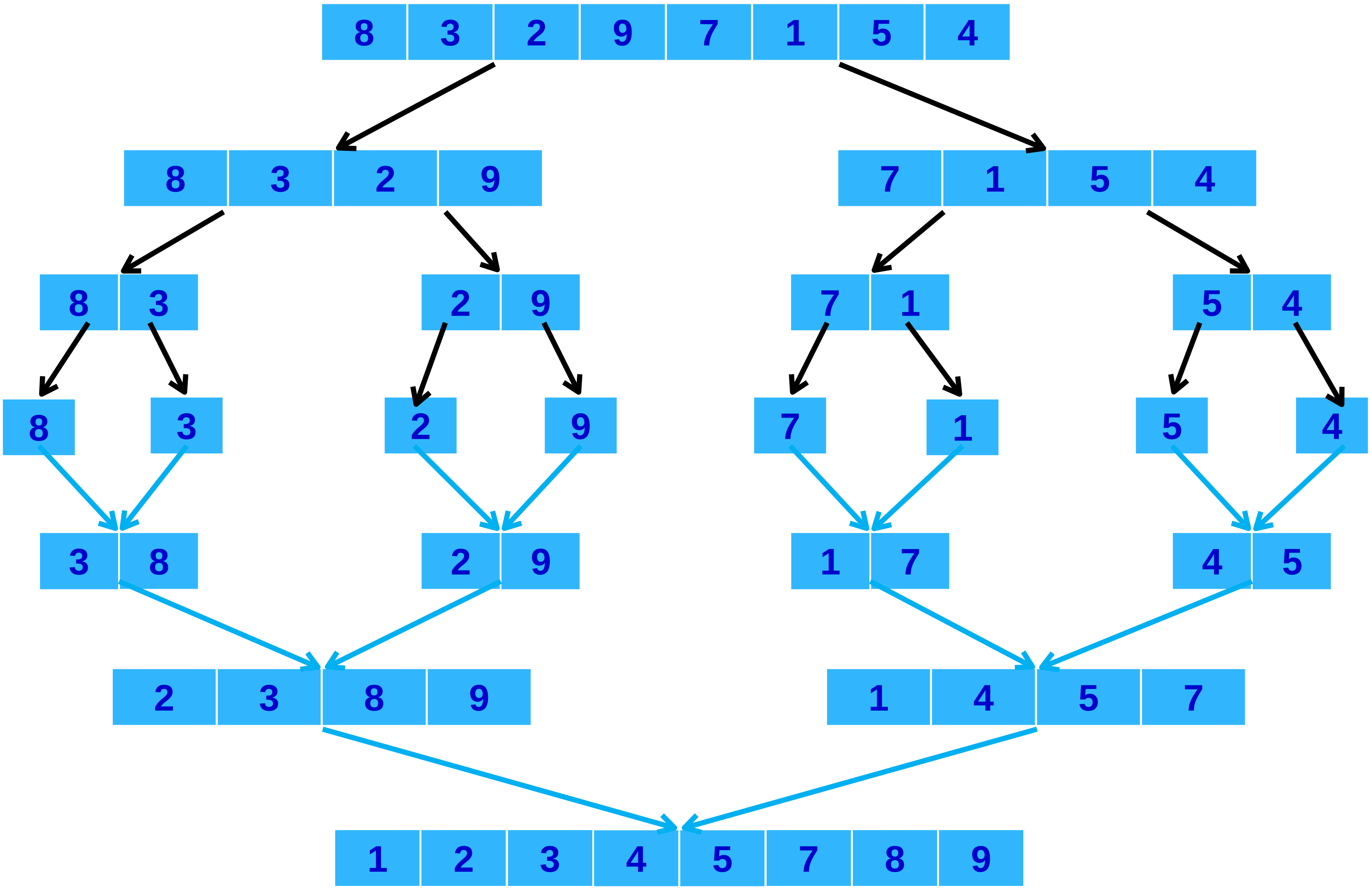
Merge Sort

- Merge-sort on an input sequence/list S with n elements consists of **three steps:**
  - **Divide:** partition S into two sequences S1 and S2 of n/2 elements each
  - **Conquer :** recursively sort S1 and S2 using merge sort
  - **Combine:** merge S1 and S2 into a unique sorted sequence

Merge Sort

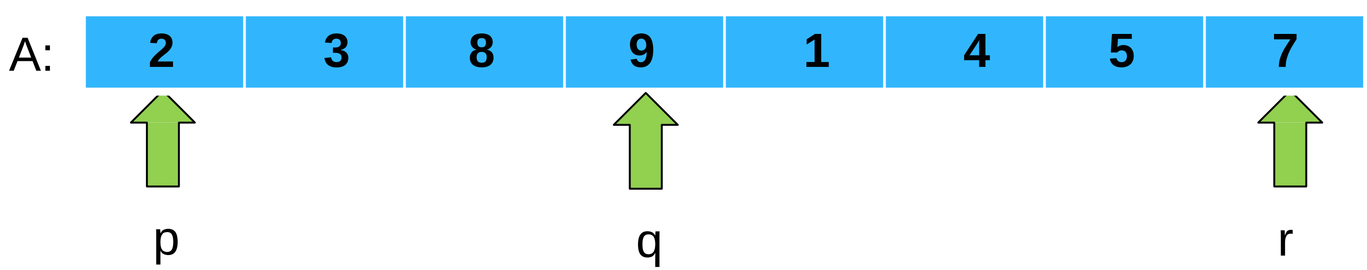
Divide: →

Conquer and Merge: →



Merge Sort: pseudo-code


```
Algorithm Merge-Sort(A, p, r):
    if p < r then
        q←⌊(p+r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
    Merge(A, p, q, r)
```



Pseudocode of the merge procedure


```
MERGE (A, p, q, r)

    n1 ← q − p + 1
    n2 ← r − q
    Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
    FOR i ← 1 TO n1 DO
            L[i] ← A[p + i − 1]
    FOR j ← 1 TO n2 DO
            R[j] ← A[q + j]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    FOR k ← p TO r DO
        IF L[i] ≤ R[ j] THEN
            A[k] ← L[i]
            i ← i + 1
        ELSE A[k] ← R[j]
            j ← j + 1
```

ANALYSIS OF MERGE SORT

ALGORITHM Mergesort(A, p, r)

IF $p < r$	
THEN $q = \text{FLOOR}[(p + r)/2]$	1
MERGE_Sort (A, p, q)	T(n/2)
MERGE_Sort (A, $q + 1$, r)	T(n/2)
MERGE (A, p, q, r)	θ(n)
	$T(n) = 2T(n/2) + \theta(n)$

Solve the Recurrence

$T(n) =$

$2T(n/2) + cn$

Use Master's Theorem:

a = 2, b = 2, $\log_2 2 = 1$

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Quick Sort

Quick Sort

- Quick-sort on an input sequence/list S with n elements consists of **three steps:**
  - **Divide:** partition S into two sequences S1 and S2 based on pivot value.
  - **Conquer :** recursively sort S1 and S2
  - **Combine:** merge S1 and S2 into a unique sorted sequence

Quick Sort

- **<u>Pivot selection</u>**
  - **There are different methods for pivot selection for instance:**
    - Use the list first element as a pivot
    - Use the list last element as a pivot
    - Use the middle element as a pivot
    - Use a random element as a pivot

Example

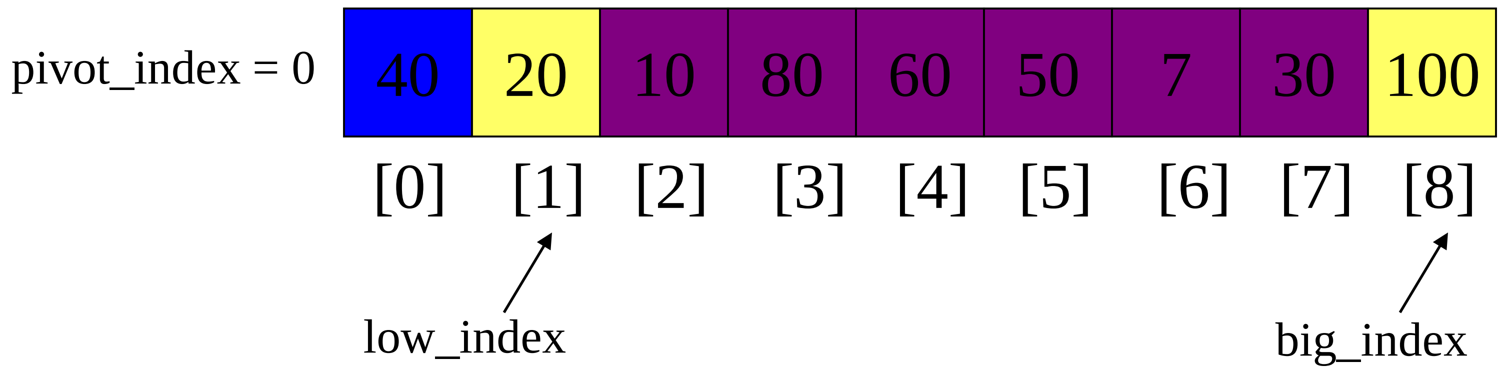
We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

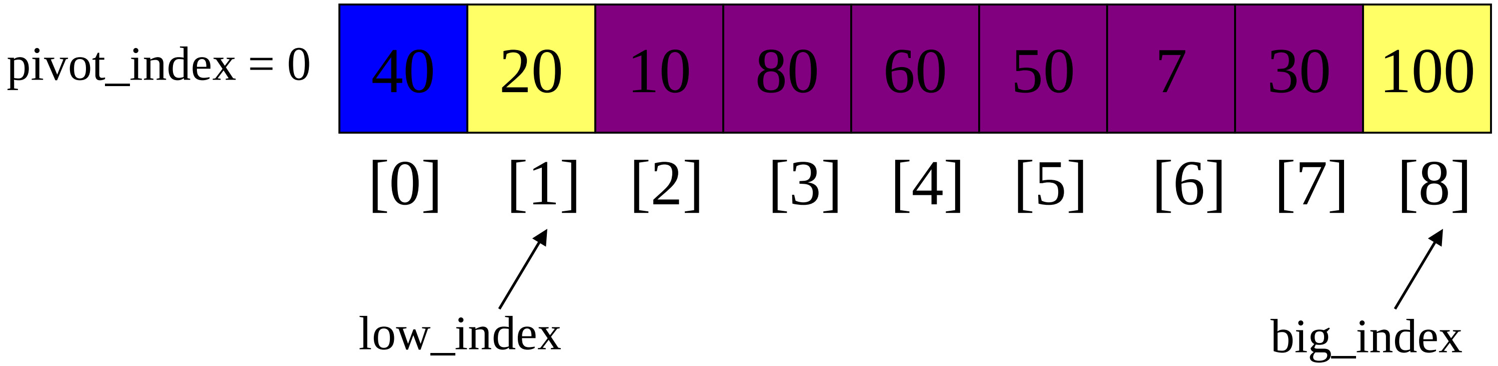
Pick Pivot Element

- There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

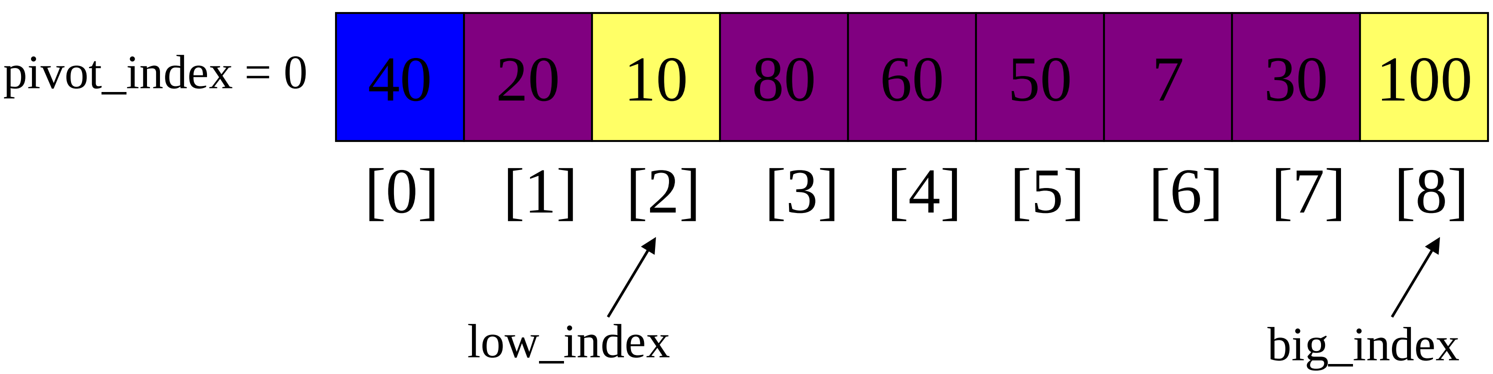
40	20	10	80	60	50	7	30	100


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
low_index
big_index

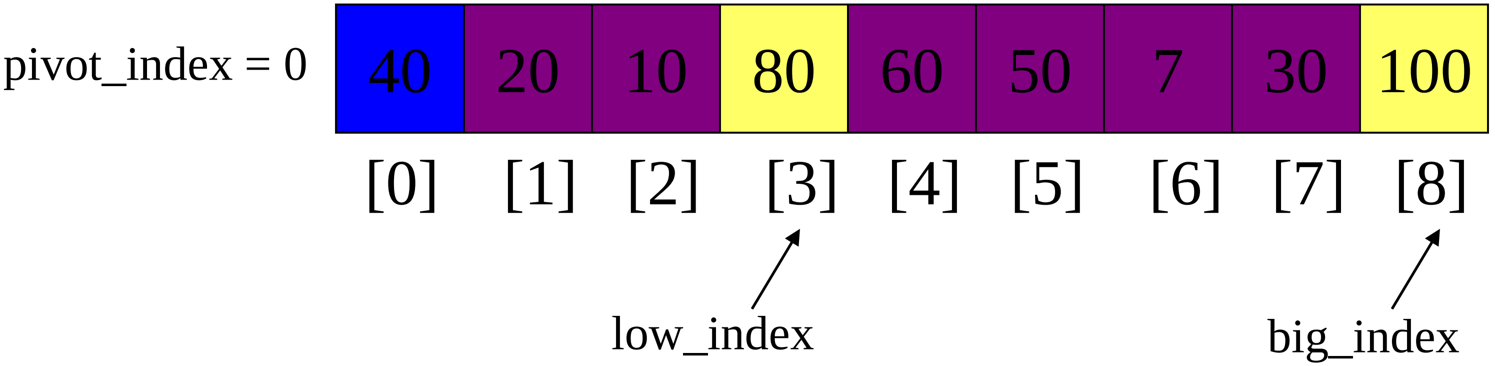
1. While arr[low_index] <= arr[pivot]
   low_index++



1. While arr[low_index] <= arr[pivot]
   low_index++



1. While arr[low_index] <= arr[pivot]
    low_index++

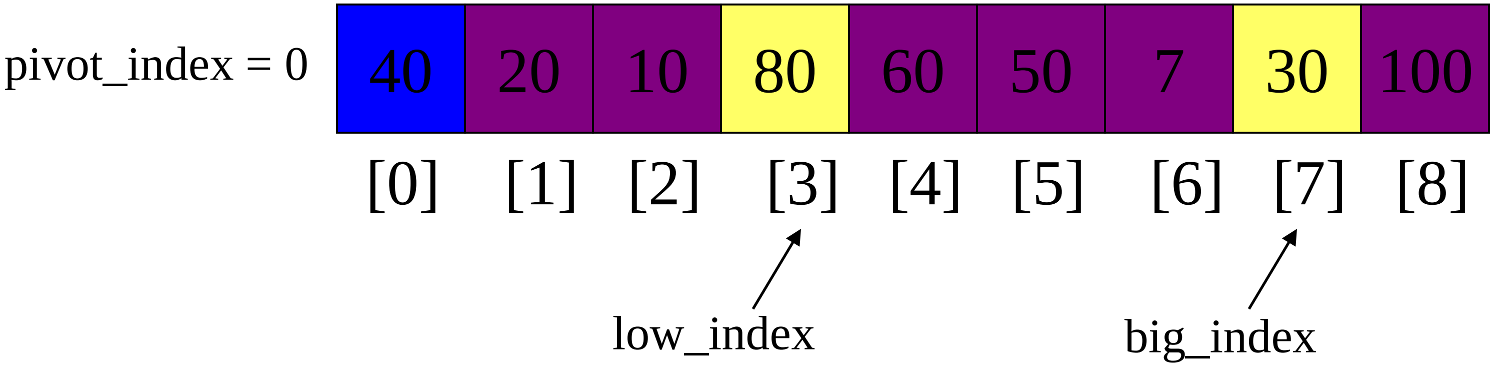


1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--

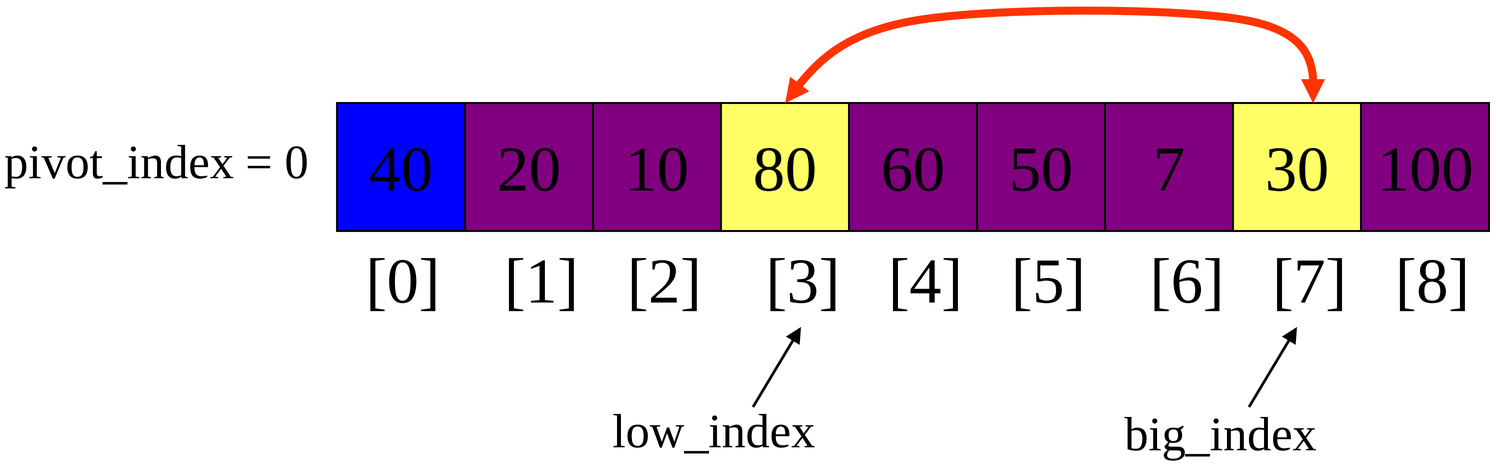
pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100
			low_index					big_index

1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]



1. While arr[low_index] <= arr[pivot]

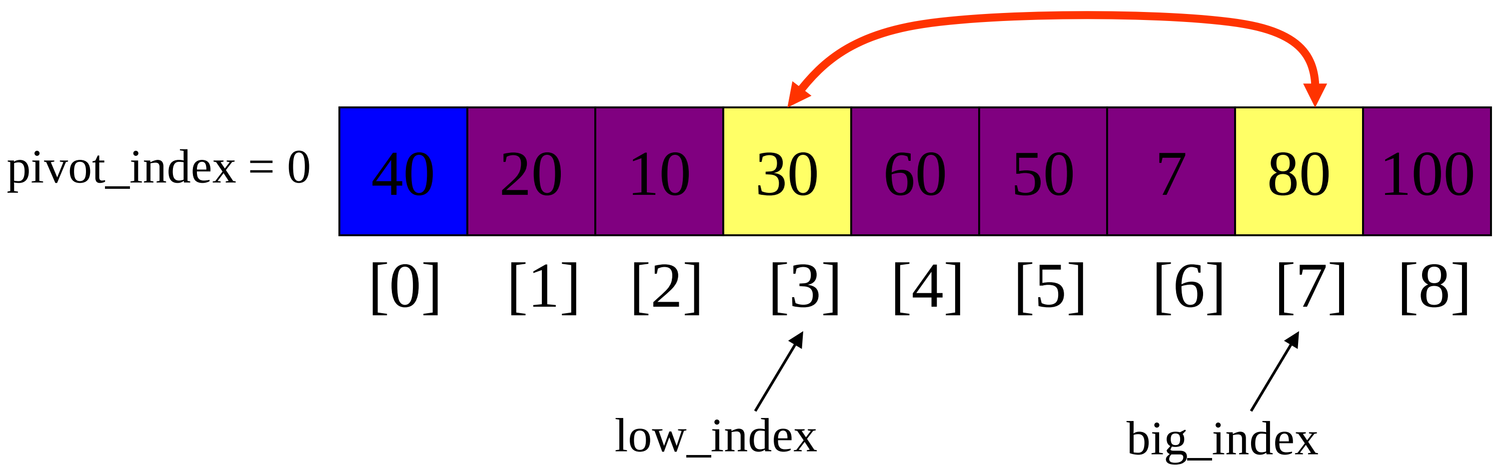
    low_index++

2. While arr[big_index] > arr[pivot]

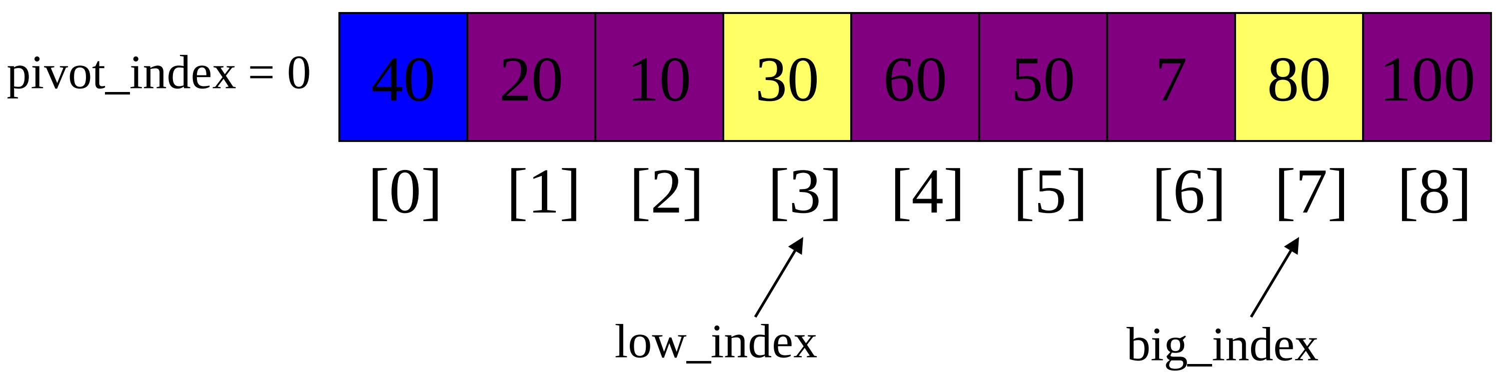
    big_index--

3. If low_index < big_index

    swap arr[low_index] and arr[big_index]



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

low_index

big_index

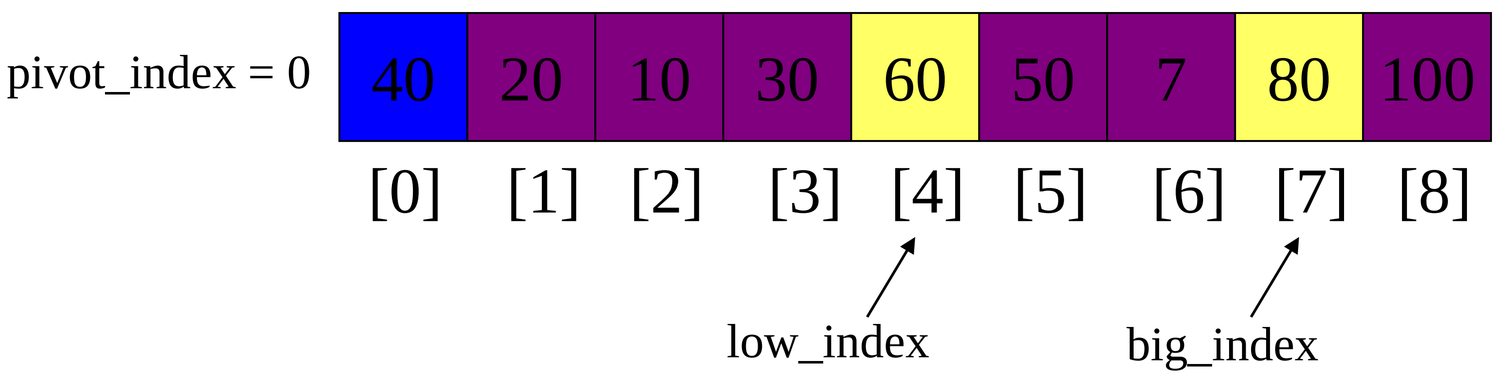
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.

pivot_index = 0

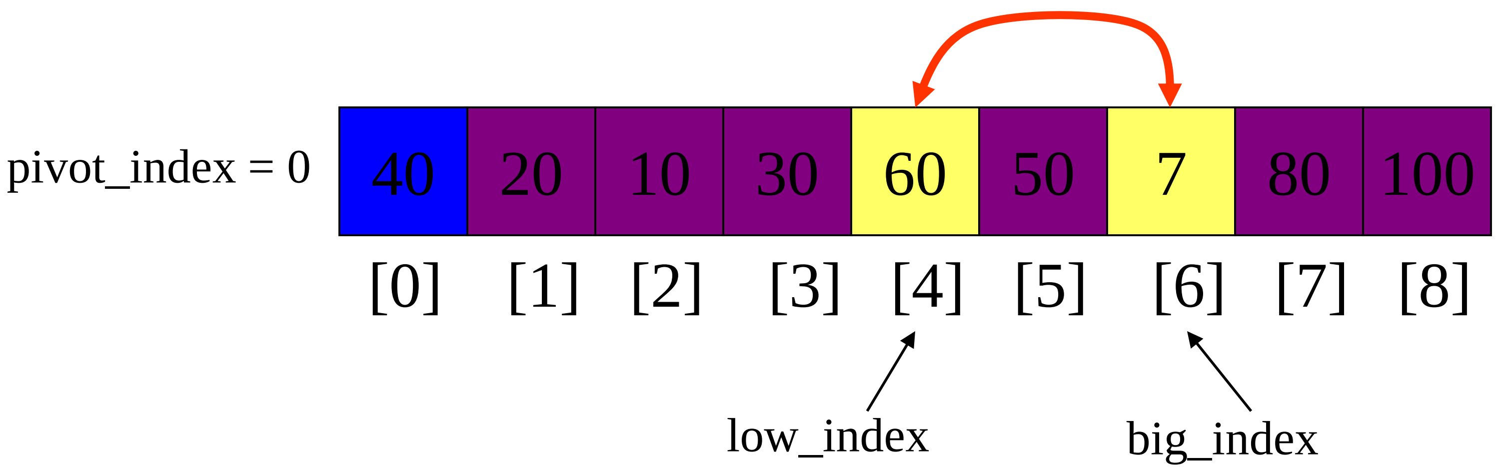
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

low_index   big_index

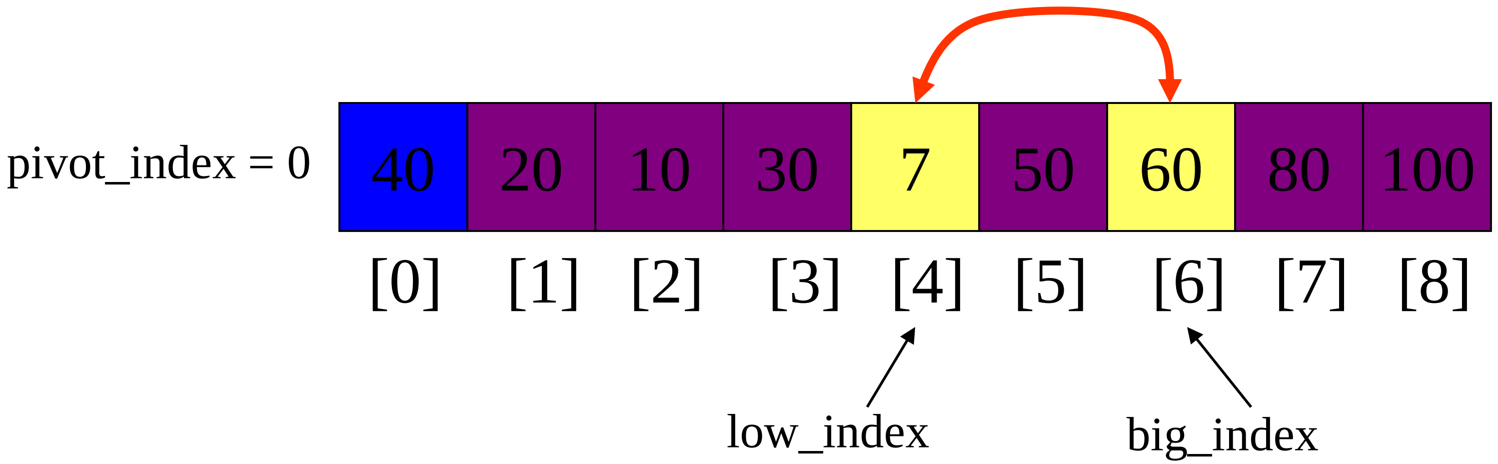
1. While arr[low_index] <= arr[pivot]
    low_index++
2. While arr[big_index] > arr[pivot]
    big_index--
3. If low_index < big_index
    swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



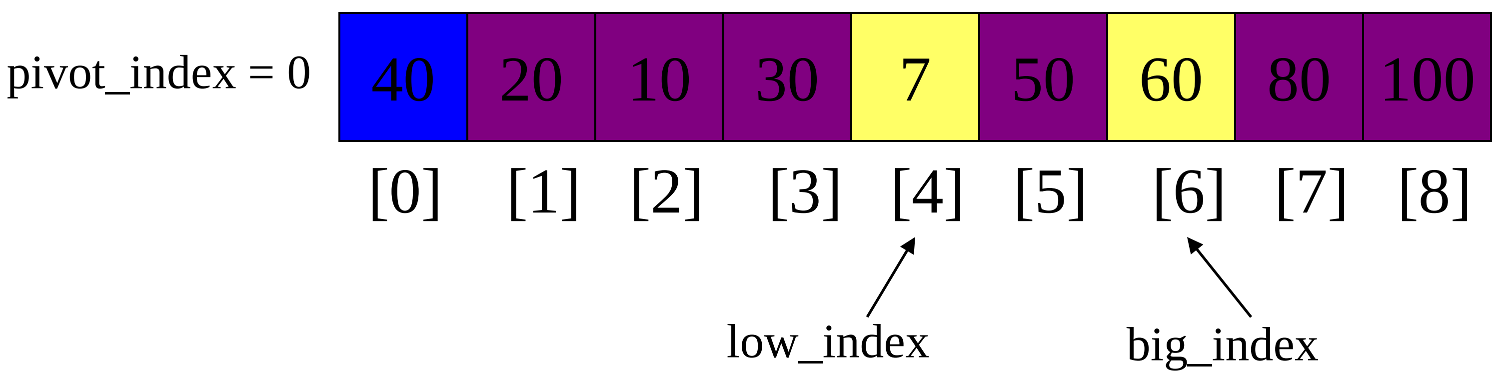
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



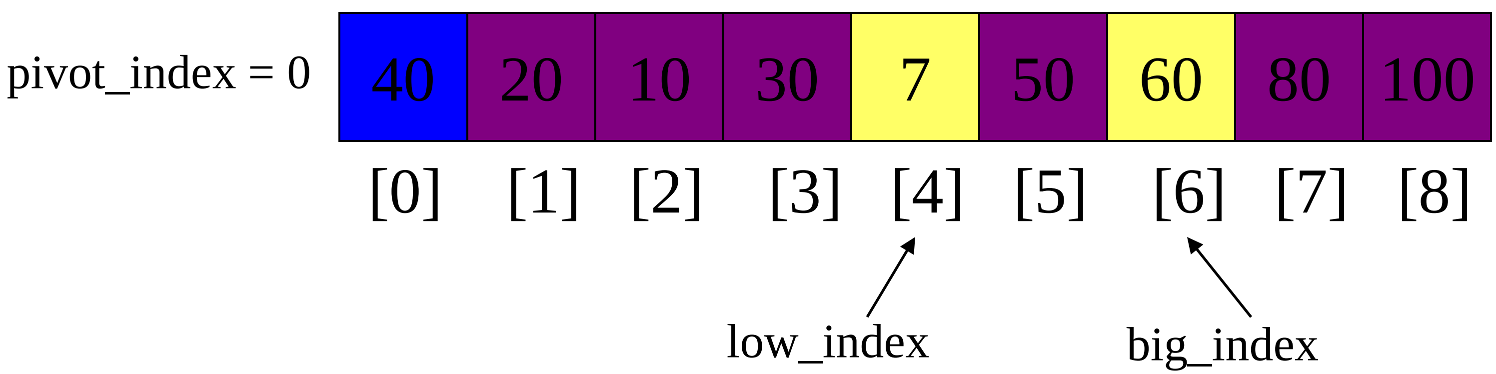
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



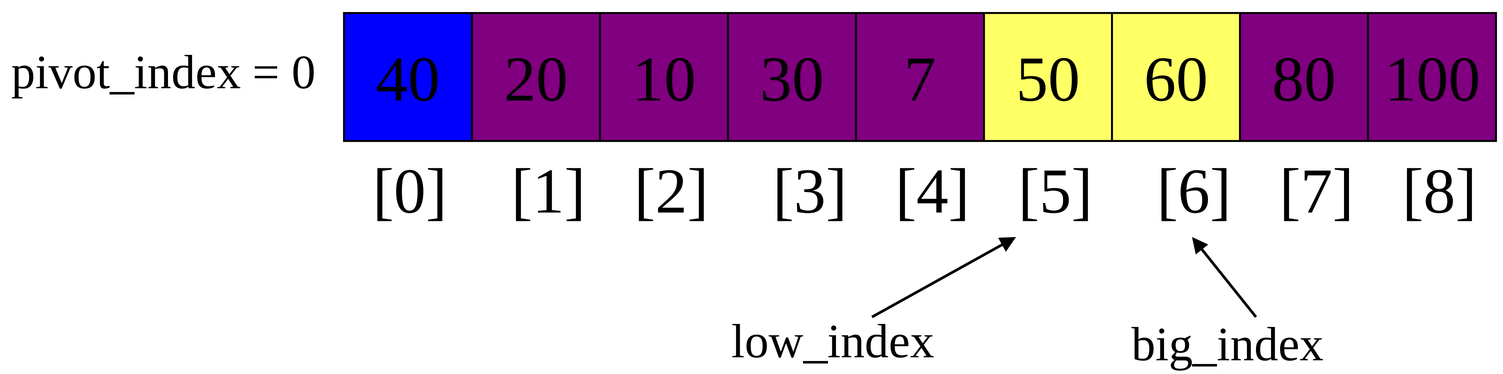
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.

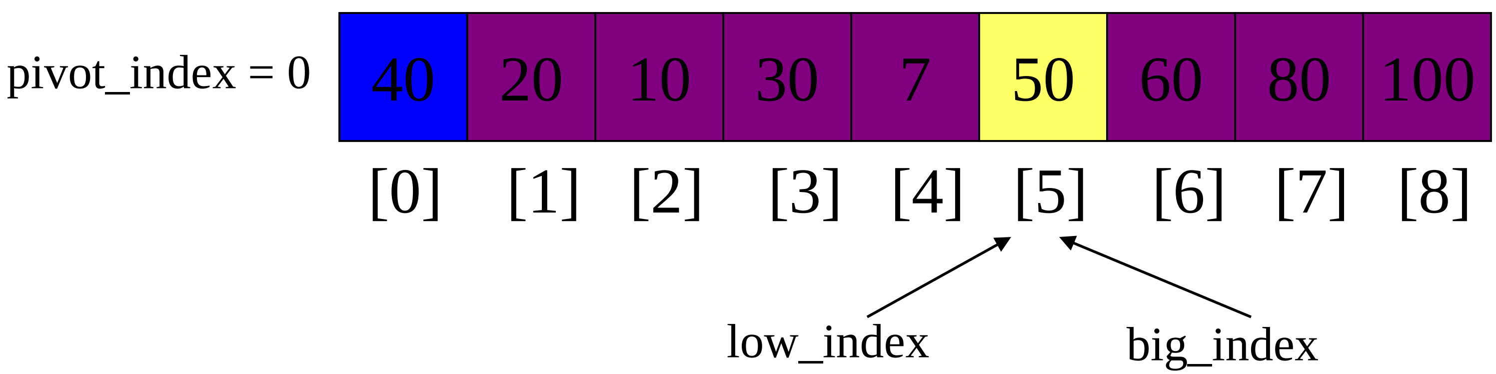
pivot_index = 0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

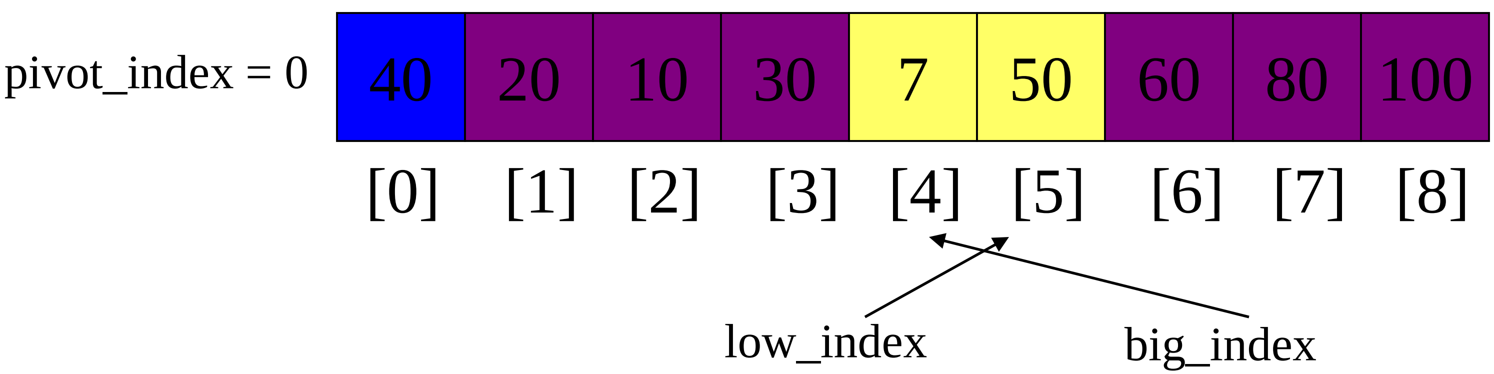
low_index

big_index

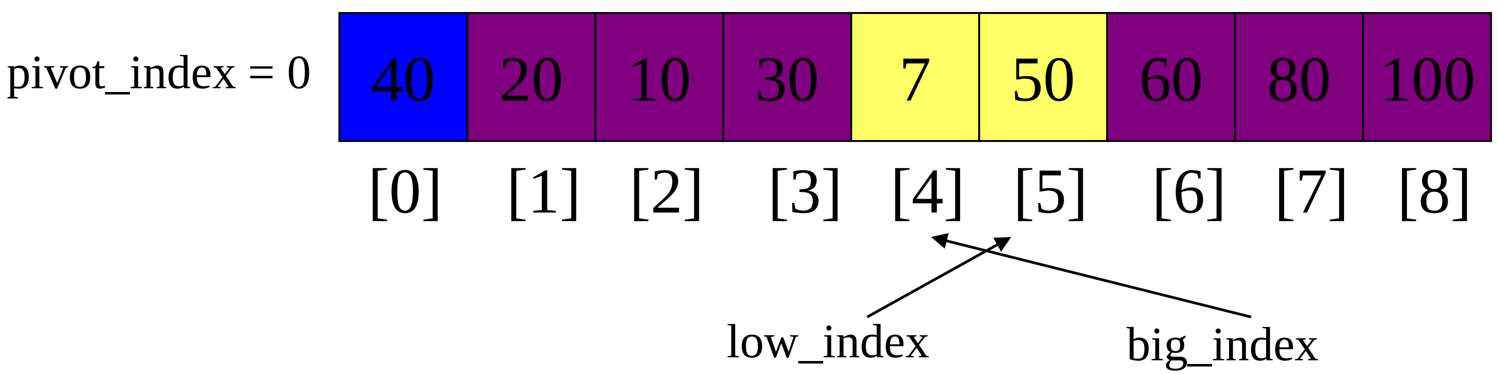
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.



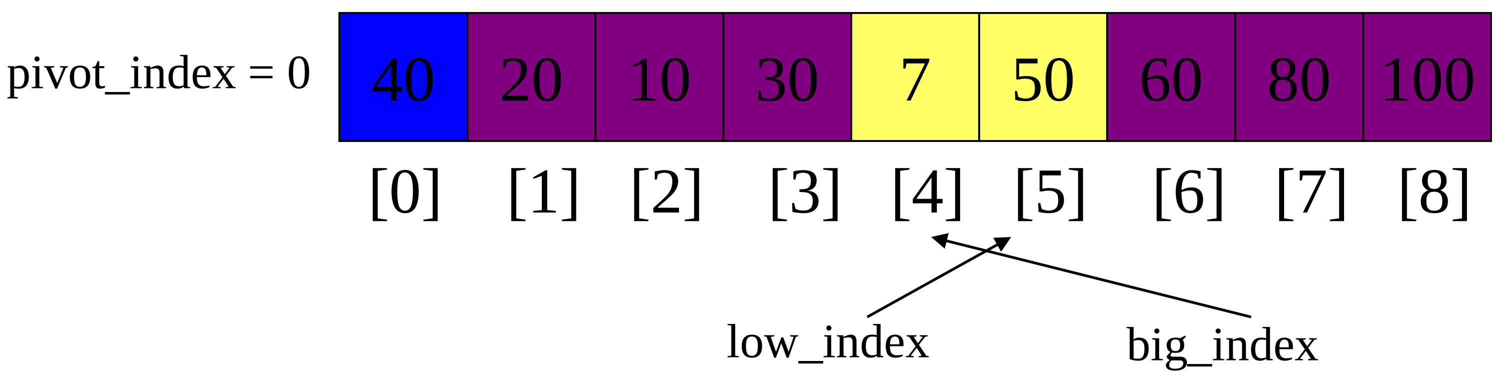
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.

pivot_index = 0

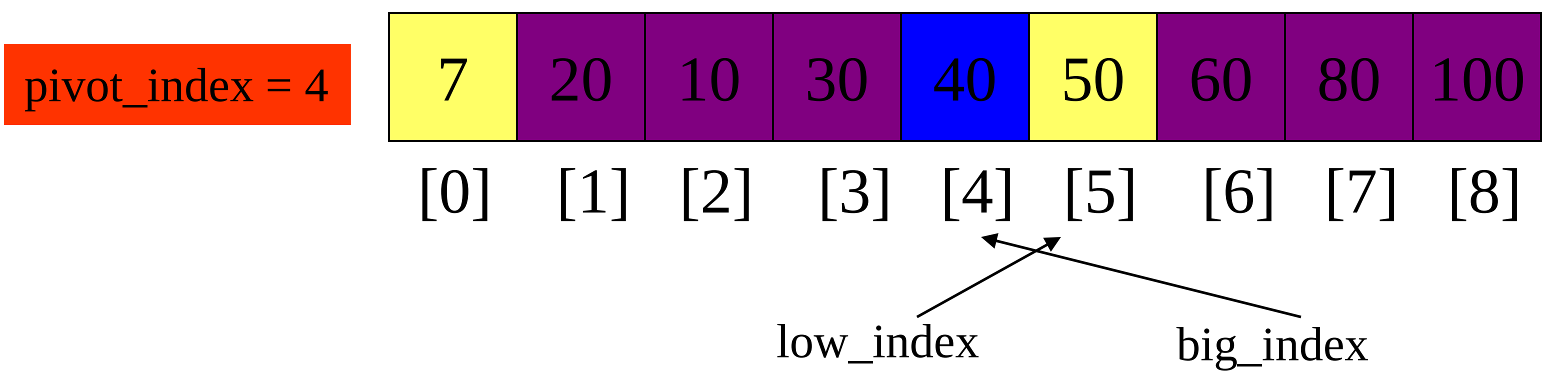
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

low_index big_index

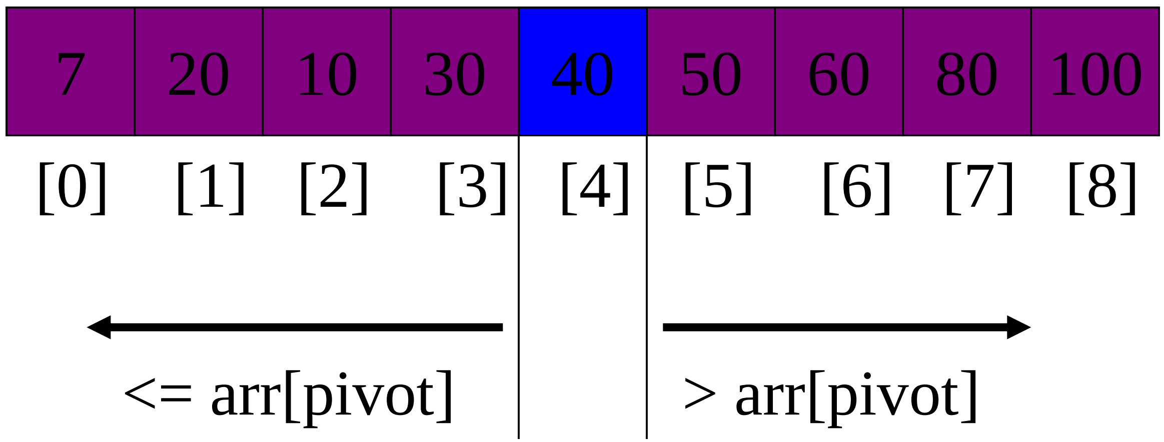
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.
5. Swap arr[big_index] and arr[pivot_index]



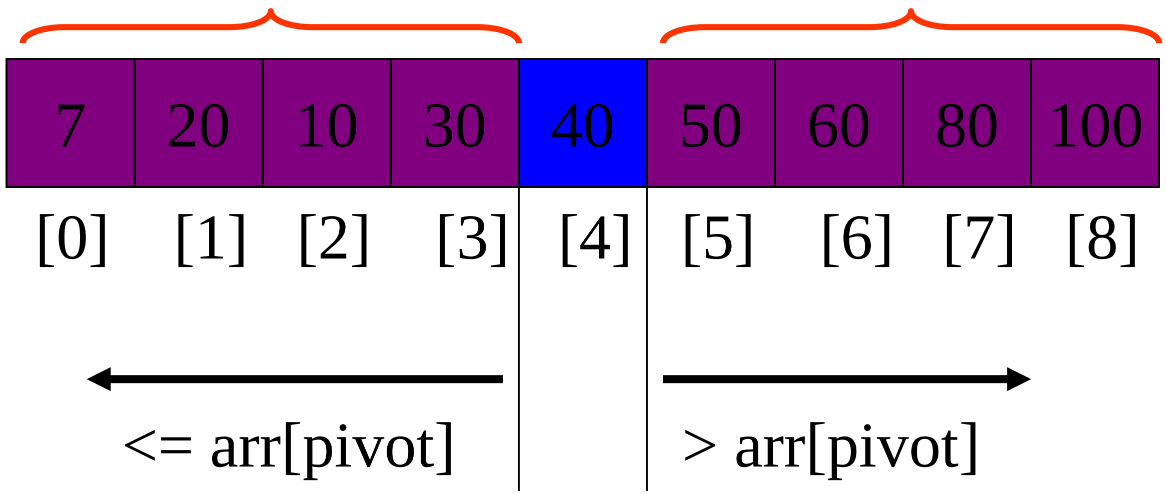
1. While arr[low_index] <= arr[pivot]
   low_index++
2. While arr[big_index] > arr[pivot]
   big_index--
3. If low_index < big_index
   swap arr[low_index] and arr[big_index]
4. While low_index <= big_index, go to 1.
5. Swap arr[big_index] and arr[pivot_index]



Partition Result



Recursion: Quicksort Sub-arrays



Quick Sort: pseud-code


```
Algorithm quicksort(A, left, right)
    if (left<right)
        pivot_index=partition(A, left, right);
        quicksort(A,left,pivot_index-1);
        quicksort(A, pivot_index+1,right);
```

Pseudocode of the partition procedure


```
Algorithm partition(A, left, right)
 low_index=left+1, big_index=right;
  p=A[left];
   while low_index <= big_index do
           while A[low_index]<=p do
                   low_index++;
            while arr[big_index]>p do
                   big_index--;
             if low_index<=big_index the
                  exchange A[low_index] ↔ A[big_index]
   End while
exchange A[left] ↔ A[big_index]
return big_index;
```

Quick-SORT Running Time

❑ Worst Case Analysis

- The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider previous partition strategy where first element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case
- Recurrence Relation:

  $T(0) = T(1) = 0$ (base case)

  $T(N) = N + T(N-1)$

Quick-SORT Running Time

❑ Worst Case Analysis

❖ Solving the RR: **Use itération (substitution) method**

T(N) = N + T(N-1)

T(N-1) = (N-1) + T(N-2)

T(N-2) = (N-2) + T(N-3)

...

T(3) = 3 + T(2)

T(2) = 2 + T(1)

T(1) = 0 **Hence,**

T(N) = N + (N-1) + (N-2) ... + 3 + 2 $\approx \frac{N^2}{2}$ which is $\mathbf{\Theta}$ **(*N^2*)**

Solve the Recurrence

❑ Best Case Analysis

- The best case occurs when the partition process always picks the middle element as pivot.
- Following is recurrence for best case.

$$T(n) = 2T(n/2) + cn$$

Use Master's Theorem:

$a = 2$, $b = 2$, $\log_2 2 = 1$

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Thanks