Sorting Algorithms

Francisco Amorós Cubells

Steps:

- Explanation Mergesort.
- Explanation Quicksort.
- Different Time complexity.
- CPU use case & Memory.
- Example with Big Data.

Mergesort – Algorithm

MERGE-SORT ALGORITHM

Input:	A list $a_1, \ldots, a_n$ of real numbers.
Output:	A permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $a_{\pi(i)} \leq a_{\pi(i+1)}$ for all $i = 1, \ldots, n-1$.

① **If** $n = 1$ **then** set $\pi(1) := 1$ and **stop** (**return** π).

② Set $m := \lfloor \frac{n}{2} \rfloor$.
Let $\rho :=$MERGE-SORT$(a_1, \ldots, a_m)$.
Let $\sigma :=$MERGE-SORT$(a_{m+1}, \ldots, a_n)$.

③ Set $k := 1$, $l := 1$.
While $k \leq m$ and $l \leq n - m$ **do**:
  If $a_{\rho(k)} \leq a_{m+\sigma(l)}$ **then** set $\pi(k + l - 1) := \rho(k)$ and $k := k + 1$
  else set $\pi(k + l - 1) := m + \sigma(l)$ and $l := l + 1$.
While $k \leq m$ **do**: Set $\pi(k + l - 1) := \rho(k)$ and $k := k + 1$.
While $l \leq n - m$ **do**: Set $\pi(k + l - 1) := m + \sigma(l)$ and $l := l + 1$.

merge of
2 sorted array

$A = [4, 7]$, $B = [1, 6, 10]$ } $C = [1]$

$A = [4, 7]$, $B = [1, 6, 10]$ } $C = [1, 4]$

$A = [4, 7]$, $B = [1, 6, 10]$ } $C = [1, 4, 6]$

$A = [4, 7]$, $B = [1, 6, 10]$ } $C = [1, 4, 6, 7]$

$A = [4, 7]$, $B = [1, 6, 10]$ } $C = [1, 4, 6, 7, 10]$

Mergesort – Algorithm

$$A = [4, 1, 7, 9, 8, 3, 6, 2, 5]$$

$$B_1 = [4, 1, 7, 9], B_2 = [8, 3, 6, 2, 5]$$

$$C_1 = [4, 1]\ C_2 = [7, 9], C_3 = [8, 3], C_4 = [6, 2, 5]$$

$$\vdots$$

$$[4], [1], [7], [9], [8], [3], [6], [2], [5]$$

Mergesort – Algorithm

[4], [1], [7], [9], [8], [3], [6], [2], [5]

[1, 4], [7, 9], [3, 8], [2, 6], [5]

[1, 4, 7, 9], [2, 3, 6, 8], [5]

[1, 2, 3, 4, 6, 7, 8, 9], [5]

[1, 2, 3, 4, 5, 6, 7, 8, 9]

Mergesort – Time Complexity

$$F(n) = \overbrace{2F(n/2)}^{\text{Recursion}} + \underbrace{n}_{\substack{\text{merge of}\\ \text{2 sorted}\\ \text{arrays}}} = O(n \cdot \log(n))$$

Quicksort – Algorithm

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quicksort – How It works?

$$A = [3, 7, 4, 2, 1, 10, 5]$$

$$p = 0, \ r = 6$$

1° $A = [\underset{p}{3}, 7, 4, 2, 1, 10, \underset{r}{5}]$

2° $A = [3, 4, 2, 1, \underset{q}{5}, 10, 7]$

3° $A = [\overset{p}{3}, 4, 2, \overset{r}{1}]$ & $[\overset{p}{10}, \overset{r}{7}]$

4° $A = [\underset{q}{1}, 4, 2, 3]$ & $[7, 10]$

5° $A = [\overset{p}{4}, 2, \overset{r}{3}]$

6° $A = [2, \underset{q}{3}, 4]$

Quicksort – How It works? – 1 Partition

partitioning(A,p,r)

$A = [3, 7, 4, 2, 1, 10, \boxed{5}] \rightarrow x$

-1

i p r

for 0 to 5 $\xrightarrow{r-1}$ $j=0$

Quicksort – How It works? – 1 Partition

partitioning(A, p, r)

$A = [3, 7, 4, 2, 1, 10, 5] \to x$

p i r

for 0 to 5 $\xrightarrow{r-1}$ j = 0

| i++

Quicksort – How It works? – 1 Partition

partitioning(A, p, r)

$A = [3, 7, 4, 2, 1, 10, 5]$ → x

p i r

for 0 to 5 $\xrightarrow{r-1}$ j = 0

i++

swap(i, j)

Quicksort – How It works? – 1 Partition

partitioning(A,p,r)

$A = [3, 7, 4, 2, 1, 10, 5]$

p i r → x

for 0 to 5 $\xrightarrow{r-1}$ j = 1

Nothing

Quicksort – How It works? – 1 Partition

partitioning(A,p,r)

A = [3, 4, 7, 2, 1, 10, 5] → x

p i r

for 0 to 5 $\xrightarrow{r-1}$ j = 2

i++

swap(i, j)

Quicksort – How It works? – 1 Partition

partitioning(A,p,r)

$A = [3, 4, 2, 7, 1, 10, 5] \to x$

p i r

for 0 to 5 $\xrightarrow{r-1}$ j = 3

i++

swap(i, j)

Quicksort – How It works? – 1 Partition

partitioning(A, p, r)

$A = [3, 4, 2, 1, 7, 10, 5] \rightarrow x$

p i r

for 0 to 5 $\xrightarrow{r-1}$ $j = 4$

- i++
- swap (i, j)

Quicksort – How It works? – 1 Partition

partitioning(A,p,r)

A = [3, 4, 2, 1, 7, 10, 5] → x

p, i, r

for 0 to 5 $\xrightarrow{r-1}$ j = 5

Nothing

Quicksort – How It works? – 1 Partition

partitioning(A,p,r)

$A = [3, 4, 2, 1, 7, 10, \boxed{5}] \to x$

p i r

swap(i(3)+1 , r)

$[3, 4, 2, 1, 5, 10, 7]$

p i r

Return (i(3)+1)

QS(A, p, i-1)   QS(A, i+1, r)

Quicksort - How It works?

$A = [3, 7, 4, 2, 1, 10, 5]$

$p = 0, r = 6$

1º $A = [\underset{p}{3}, 7, 4, 2, 1, 10, \underset{r}{5}]$

2º $A = [3, 4, 2, 1, \underset{q}{5}, 10, 7]$

3º $A = [\overset{p}{3}, 4, 2, \overset{r}{1}]$ & $[\overset{p}{10}, \overset{r}{7}]$

4º $A = [\underset{q}{1}, 4, 2, 3]$ & $[7, 10]$

5º $A = [\overset{p}{4}, 2, \overset{r}{3}]$

6º $A = [2, \underset{q}{3}, 4]$

Quicksort – Time Complexity

Different Types of time Complexity:

- Best
- Worst
- Average

Introduction:

Time Complexity

$O(1) = f(n)\{2 + n[3]\}$

$O(n) = f(n)\{$ for element in n:
sum += element $\}$

$O(n^2) = f(n)\{$ for element in n:
for item in n:
mult += element * item $\}$

Quicksort – Time Complexity – *Best*

O(n) Best Case:

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

$$F(n) = 2F\left(\frac{n}{2}\right) + n$$

Quicksort – Time Complexity – *Best*

O(n) Best Case:

QUICKSORT(A, p, r)

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

$$F(n) = 2F\left(\frac{n}{2}\right) + n$$

$$O(n \cdot \log_2(n))$$

Quicksort – Time Complexity – *Worse*

O(n) Worse Case:

$$F(n) = F(n-1) + F(0) + n$$

$$O(n^2)$$

Quicksort – Time Complexity – *Average*

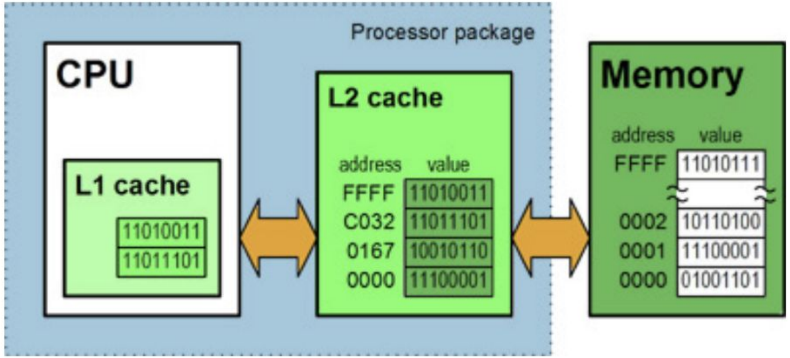
O(n) Average Case: (Unbalanced)

$$F(n) = F\left(\frac{9n}{10}\right) + F\left(\frac{n}{10}\right) + n$$

$$O(n \cdot \log_2(n))$$

CPU use case & Memory

Cache:



Spatial Locality :

WIl those instructions which are stored nearby to the recently executed instruction have high chances of execution. (Sequentially accessed data).

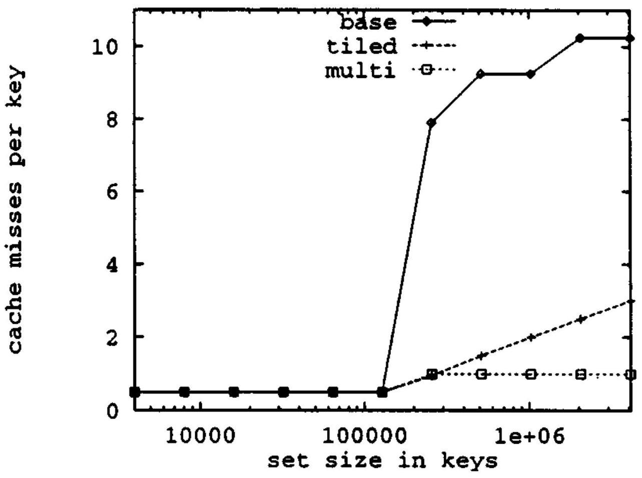
Temporal Locality :

A instruction which is recently executed have high chances of execution again.
So the instruction is kept in cache memory such that it can be fetched easily and takes no time in searching for the same instruction.

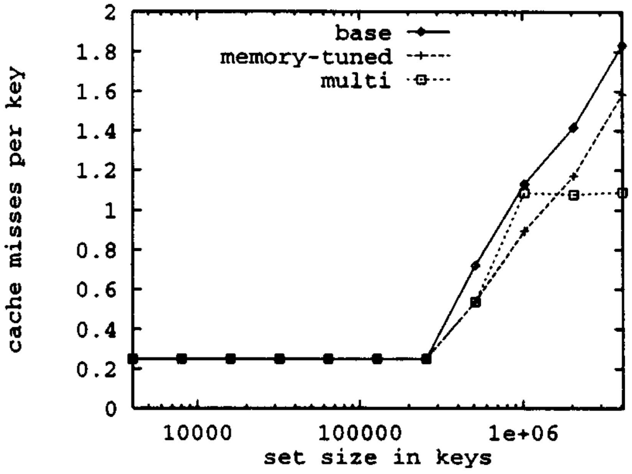
CPU use case & Memory (Paper)

Cache:

Merge-sort



Quick-sort



CPU use case & Memory (Big - Data)