FTP_Alg
Quick Sort and Counting Sort

jungkyu.canci@hslu.ch

HS2024

Quicksort

Divide

Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

Conquer

Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine

Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

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

Quicksort

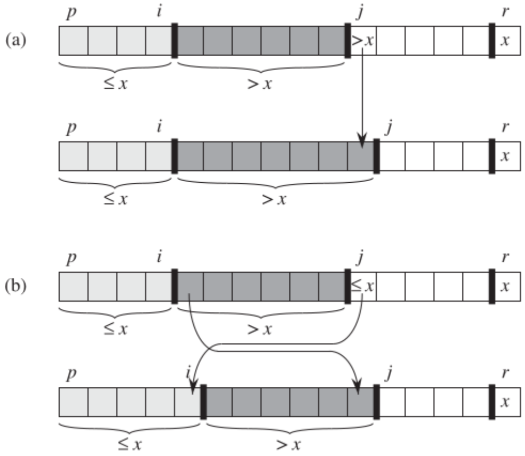
The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

In this version of quicksort we have $q = r$. Lines 3–6, the following invariant must hold true:

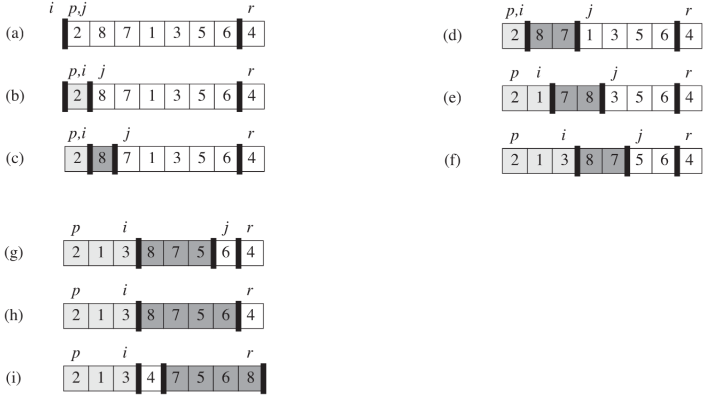
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Quicksort: the two cases of PARTITION



- if $A[j] > x$: increment j
- if $A[j] \leq x$: increments i, swap $A[i]$ and $A[j]$, increments j.

Quicksort Step by Step



Worst-Case Partitioning

We have the worst case, when at each partitioning of n elements, the subroutine produces one subproblem with $n-1$ elements and one with 0 elements. Since the partitioning costs $\Theta(n)$, we have

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

One can see (Exercise) that $T(n) = \Theta(n^2)$.

Best-Case Partitioning

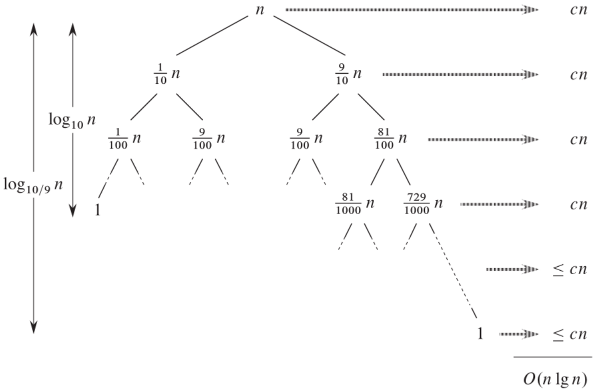
We have the best case, when at each partitioning the two subproblems have no more than $n/2$ elements (by partitioning n elements). In this case we have

$$T(n) = 2T(n/2) + \Theta(n)$$

and we have already seen that in this case we have $T(n) \in O(n \log n)$.

Balanced Partitioning

We present this following example where the partition algorithm produces a 9–to–1 proportional split.

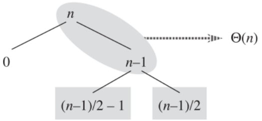


From this pictures one sees that $T(n) = O(n \log n)$. The same reasoning can be also done for a 99–to–1 proportional partitioning!

Intuition for Average-Case

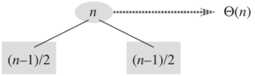
In average partititon produces "bad" and "good" splits.

Bad split



(a)

Good split



(b)

Suppose that good and bad split alternate. A bad split followed by a good split produce three subarrays of size 0, $\frac{n-1}{2} - 1$ and $\frac{n-1}{2}$ so the combined partition cost is

$$\Theta(0) + \Theta\left(\frac{n-1}{2} - 1\right) + \Theta\left(\frac{n-1}{2}\right) = \Theta(n).$$

We can say that in average "good splits compensate bad splits".

A Randomized Version of Quicksort

Instead of always taking as pivot in quicksort the element $A[r]$, we could randomize the process by calling a random choice of the index i for choosing the pivot $A[q] = A[i]$.

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

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

One can see that we have the worst case, when the randomly chosen i is such that $A[i]$ is the highest value in the array and so the partitioning produce a split with a $n-1$ subproblem. As we have seen if this happens to each iteration, the running time is $\Theta(n^2)$.

Expected running time

- We could repeat the same reasoning that we have considered in the slide “Intuition for Average-Case”.
- If we had only good splits the height of the tree of the subproblems would be $O(\log n)$, where at each level we will have cost $\Theta(n)$.
- Now in a “generic”situation we could assume that we add some extremely unbalanced split among the good ones.
- By considering the block of balanced split with the successive unbalance we will have something, which costs $\Theta(n)$, and we will have at most $O(\log n)$ of such blocks.
- Thus the expected running time is $\Theta(n \log n)$. One can prove it by using the expected value on the random variable **running time** (e.g. see the book "Introduction to Algorithms"...)
- One can calculate explicitly the constants coming out in $T(n)$ in the expected case and see that they are quite small in comparison with merge sort.

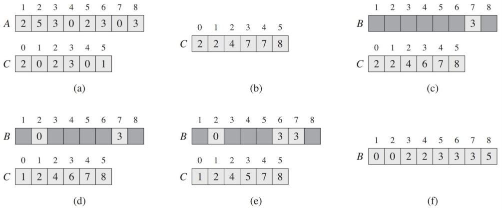
Sorting in linear time

- So far we have revised some sorting algorithms that use some comparison arguments for sorting the objects (merge sort, heapsort, quicksort).
- We have seen that in all these above procedures, there exist cases where the running time is $\Theta(n)$, with n the number of elements to be sorted.
- It is possible to prove that every sorting algorithms that use comparison methods, there exist cases where the running time is $\Theta(n)$.
- There exist some sorting algorithms, that run in linear time, i.e. in $O(n)$, where we use some extra information for sorting problem. Here we present **counting sort**.
- We apply counting sort, when we know that the inputs elements are n integers between 0 and k, for a known positive integer k.
- When $k = O(n)$, counting sort runs in $\Theta(n)$ time.

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

The procedure starts by creating an array C that tell us the number of elements having key i for each $0 \leq i \leq k$. The step after is to change C, which now tells us the number of elements having key $\leq i$ for each $0 \leq i \leq k$.

Then the procedure creates the array B containing the sorted list by using the info in C.



Counting Sort: Running Time

- The **for** loop of lines 2-3 takes $\Theta(k)$ time.
- The **for** loop of lines 4-5 takes $\Theta(n)$ time.
- The **for** loop of lines 7-8 takes $\Theta(k)$ time.
- The **for** loop of lines 10-12 takes $\Theta(n)$ time.
- The other lines are comments (or the input), thus the running time is $\Theta(k + n)$.
- If the procedure is applied with $k = \Theta(n)$, the running time is $\Theta(n)$ (so in linear time with respect n).