Design and Analysis of Algorithms

Quick Sort

Lecture 11-12

Instructor: Dr. G P Gupta

Performance

- A triumph of analysis by C.A.R. Hoare
- Worst-case execution time – $\Theta(n^2)$.
- Average-case execution time – $\Theta(n \lg n)$.
  - » How do the above compare with the complexities of other sorting algorithms?
- Empirical and analytical studies show that quicksort can be ***expected*** to be twice as fast as its competitors.

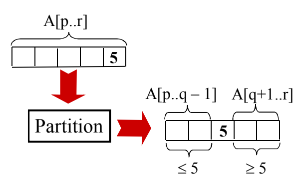
Design

- **Follows the divide-and-conquer paradigm.**
- ***Divide***: Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q–1]$ and $A[q+1..r]$.
  - » Each element in $A[p..q–1] \le A[q]$.
  - » $A[q] \le$ each element in $A[q+1..r]$.
  - » Index q is computed as part of the partitioning procedure.
- ***Conquer***: Sort the two subarrays by recursive calls to quicksort.
- ***Combine***: The subarrays are sorted in place – no work is needed to combine them.
- How do the divide and combine steps of quicksort compare with those of merge sort?

Pseudocode

```
Quicksort(A, p, r)
   if p < r then
      q := Partition(A, p, r);
      Quicksort(A, p, q – 1);
      Quicksort(A, q + 1, r)
   fi
```

```
Partition(A, p, r)
   x, i := A[r], p – 1;
   for j := p to r – 1 do
      if A[j] ≤ x then
         i := i + 1;
         A[i] ↔ A[j]
      fi
   od;
   A[i + 1] ↔ A[r];
   return i + 1
```



Example

```
                     p                     r
initially:           2 5 8 3 9 4 1 7 10 6       note: pivot (x) = 6
                   i j

next iteration:      2 5 8 3 9 4 1 7 10 6
                     i j

next iteration:      2 5 8 3 9 4 1 7 10 6
                       i j

next iteration:      2 5 8 3 9 4 1 7 10 6
                       i   j

next iteration:      2 5 3 8 9 4 1 7 10 6
                         i   j
```

```
Partition(A, p, r)
   x = A[r] ; i = p – 1;
   for j := p to r – 1 do
      if A[j] ≤ x then
         i := i + 1;
         A[i] ↔ A[j]

   A[i + 1] ↔ A[r];
   return i + 1
```

Example (Continued)

```
next iteration:      2 5 3 8 9 4 1 7 10 6
                         i   j

next iteration:      2 5 3 8 9 4 1 7 10 6
                         i     j

next iteration:      2 5 3 4 9 8 1 7 10 6
                           i     j

next iteration:      2 5 3 4 1 8 9 7 10 6
                             i     j

next iteration:      2 5 3 4 1 8 9 7 10 6
                             i       j

next iteration:      2 5 3 4 1 8 9 7 10 6
                             i          j

after final swap:    2 5 3 4 1 6 9 7 10 8
                             i          j
```

```
Partition(A, p, r)
   x= A[r]; i = p – 1;
   for j := p to r – 1 do
      if A[j] ≤ x then
         i := i + 1;
         A[i] ↔ A[j]

   A[i + 1] ↔ A[r];
   return i + 1
```

Partitioning

- ***Select the last element*** A[r] in the subarray $A[p..r]$ as the ***pivot*** – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.

1. *$A[p..i]$* — All entries in this region are ≤ ***pivot***.
2. *$A[i+1..j-1]$* — All entries in this region are > ***pivot***.
3. *$A[r]$ = pivot*.
4. $A[j..r-1]$ — Not known how they compare to *pivot*.

- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the LI.)

Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- $\Theta(n)$: $n = r - p + 1$.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Exercise

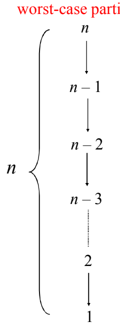
- illustrate the operation of PARTITION on the array
  A ={13; 19; 9; 5; 12; 8; 7; 4; 21; 2; 6; 11}
- What value of q does PARTITION return when all elements in the array *A[p..r]* have the same value?

Performance of quicksort

- **Running time of quicksort depends on whether the *partitioning is balanced or not.***
- **Worst-Case Partitioning (Unbalanced Partitions):**
  - » Occurs when every call to partition results in the most **unbalanced partition**.
  - » ***Partition is most unbalanced when***
    - Subproblem 1 is of size $n - 1$, and subproblem 2 is of size 0 or vice versa.
    - $pivot \geq$ every element in $A[p..r-1]$ or $pivot <$ every element in $A[p..r-1]$.
  - » ***Every call to partition is most unbalanced when***
    - Array $A[1..n]$ is sorted or reverse sorted!

Worst-case Partition Analysis

Recursion tree for worst-case partition



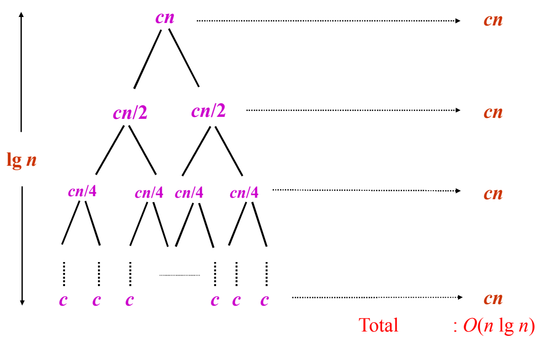
Running time for worst-case partitions at each recursive level:

$$T(n) = T(n-1) + T(0) + \text{PartitionTime}(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \sum_{k=1 \text{ to } n} \Theta(k)$$
$$= \Theta(\sum_{k=1 \text{ to } n} k)$$
$$= \Theta(n^2)$$

Best-case Partitioning

- **Size of each subproblem ≤ $n/2$.**
  - » One of the subproblems is of size $\lfloor n/2 \rfloor$
  - » The other is of size $\lceil n/2 \rceil - 1$.
- **Recurrence for running time**
  - » $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$
    $= 2T(n/2) + \Theta(n)$
- $\boldsymbol{T(n) = \Theta(n \lg n)}$

Recursion Tree for Best-case Partition



Variations

- Quicksort is not very efficient on small lists.
- This is a problem because Quicksort will be called on lots of small lists.
- **Fix 1:** Use Insertion Sort on small problems.
- **Fix 2:** Leave small problems unsorted. Fix with one final Insertion Sort at end.
  - » **Note:** *Insertion Sort is very fast on almost-sorted lists.*

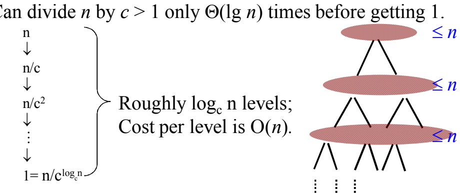
Unbalanced Partition Analysis

What happens if we get ***poorly-balanced partitions***,

e.g., something like: $T(n) \le T(9n/10) + T(n/10) + \Theta(n)$?

Still get $\Theta(n \lg n)$!! (As long as the split is of constant proportionality.)

Intuition: Can divide n by $c > 1$ only $\Theta(\lg n)$ times before getting 1.



(**Remember:** Different base logs are related by a constant.)

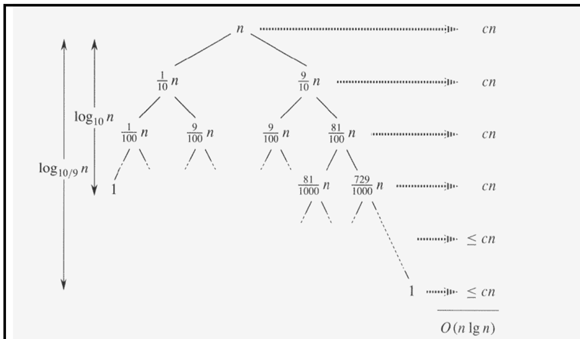


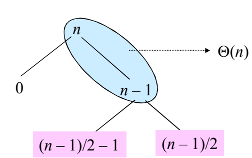
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Intuition for the Average Case

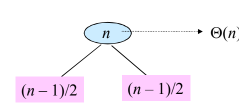
- Partitioning is unlikely to happen in the same way at every level.
  - » Split ratio is different for different levels.

    (Contrary to our assumption in the previous slide.)
- Partition produces a mix of "good" and "bad" splits, distributed randomly in the recursion tree.
- ***What is the running time likely to be in such a case?***

Intuition for the Average Case



Bad split followed by a good split:
Produces subarrays of sizes 0, $(n-1)/2 - 1$, and $(n-1)/2$.
Cost of partitioning :
$\Theta(n) + \Theta(n\text{-}1) = \Theta(n)$.



Good split at the first level:
Produces two subarrays of size $(n-1)/2$.
Cost of partitioning :
$\Theta(n)$.

- Situation at the end of case 1 is not worse than that at the end of case 2.
- ***When splits alternate between good and bad***, the cost of bad split can be absorbed into the cost of good split.
- **Thus, running time is $O(n \lg n)$, though with larger hidden constants.**

Randomized Quicksort

- ***Want to make running time independent of input ordering.***
- **How can we do that?**
  - » Make the algorithm randomized.
  - » Make ***every possible input equally likely***.
    - Can randomly shuffle to permute the entire array.
    - For quicksort, it is sufficient if we can ensure that every element is equally likely to be the *pivot*.
    - So, we choose an element in $A[p..r]$ and exchange it with $A[r]$.
    - Because the *pivot* is randomly chosen, we expect the partitioning to be well balanced on average.

Randomized Version

Want to make running time independent of input ordering.

```
Randomized-Partition(A, p, r)
    i := Random(p, r);
    A[r] ↔ A[i];
    Partition(A, p, r)
```

```
Randomized-Quicksort(A, p, r)
    if p < r then
        q := Randomized-Partition(A, p, r);
        Randomized-Quicksort(A, p, q – 1);
        Randomized-Quicksort(A, q + 1, r)
    fi
```