Chapter 7: Quick Sort

**Quick Sort**

Partition set into two using randomly chosen pivot

sort the first half.

sort the second half.
Quick Sort

Glue pieces together.
(No real work)

Quick Sort

Quicksort

Quicksort advantages:
- Sorts in place
- Sorts $O(n \lg n)$ in the average case
- Very efficient in practice

Quicksort disadvantages:
- Sorts $O(n^2)$ in the worst case
- Not stable
- Does not preserve the relative order of elements with equal keys
- Sorting algorithm (stable) if 2 records with same key stay in original order
- But in practice, it’s quick
- And the worst case doesn’t happen often … sorted

Another divide-and-conquer algorithm:
- Divide: $A[p…r]$ is partitioned (rearranged) into two nonempty subarrays $A[p…q-1]$ and $A[q+1…r]$ s.t. each element of $A[p…q-1]$ is less than or equal to each element of $A[q+1…r]$. Index $q$ is computed here, called pivot.
- Conquer: two subarrays are sorted by recursive calls to quicksort.
- Combine: unlike merge sort, no work needed since the subarrays are sorted in place already.

The basic algorithm to sort an array $A$ consists of the following four easy steps:
- If the number of elements in $A$ is 0 or 1, then return
- Pick any element $v$ in $A$. This is called the pivot
- Partition $A-\{v\}$ (the remaining elements in $A$) into two disjoint groups:
  - $A_1 = \{x \in A-\{v\} | x \leq v\}$ and
  - $A_2 = \{x \in A-\{v\} | x \geq v\}$
- return
  - $\{\text{quicksort}(A_1) \text{ followed by } v \text{ followed by } \text{quicksort}(A_2)\}$
**Quicksort**
- Small instance has \( n \leq 1 \)
  - Every small instance is a sorted instance
- To sort a large instance:
  - select a pivot element from out of the \( n \) elements
- Partition the \( n \) elements into 3 groups left, middle and right
  - The middle group contains only the pivot element
  - All elements in the left group are \( \leq \) pivot
  - All elements in the right group are \( \geq \) pivot
- Sort left and right groups recursively
- Answer is sorted left group, followed by middle group followed by sorted right group

**Example**

\[ 6 \quad 2 \quad 8 \quad 5 \quad 11 \quad 10 \quad 4 \quad 1 \quad 9 \quad 7 \quad 3 \]

Use 6 as the pivot

\[ 2 \quad 5 \quad 4 \quad 1 \quad 3 \quad 6 \quad 7 \quad 9 \quad 10 \quad 11 \quad 8 \]

Sort left and right groups recursively

**Quicksort Code**

```c
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r)
        Quicksort(A, p , q-1)
        Quicksort(A, q+1 , r)
    }
}
```

- Initial call is \( \text{Quicksort}(A, 1, n) \), where \( n \) in the length of \( A \)

**Partition**
- Clearly, all the action takes place in the \text{partition()}\ function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray \( \leq \) all values in second
  - Returns the index of the “pivot” element separating the two subarrays
Partition Code

```c
Partition(A, p, r) {
    x = A[r] // x is pivot
    i = p - 1
    for j = p to r - 1 {
        do if A[j] <= x 
            then 
                i = i + 1
        } 
    return i+1  // partition() runs in O(n) time
}
```

Partition Example

\[ A = \{2, 8, 7, 1, 3, 5, 6, 4\} \]

13

Partition Example Explanation

- **Red** shaded elements are in the first partition with values \( \leq x \) (pivot)
- **Gray** shaded elements are in the second partition with values \( \geq x \) (pivot)
- The unshaded elements have no yet been put in one of the first two partitions
- The final white element is the pivot

Choice Of Pivot

- Pivot is the **rightmost** element in list that is to be sorted
  - Textbook implementation does this
- **Randomly** select one of the elements to be sorted as the pivot
  - When sorting \( A[6:20] \), generate a random number \( r \) in the range \([6, 20]\)
  - Use \( A[i] \) as the pivot
Choice Of Pivot

- **Median-of-Three** rule - from the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot.
  - Select the element with median (i.e., middle) key.

When the pivot is picked at random or when the median-of-three rule is used, we can use the quicksort code of the *textbook provided*, we first swap the rightmost element and the chosen pivot.

Runtime of Quicksort

- **Worst case:**
  - every time nothing to move
  - pivot = left (right) end of subarray
  - $\Theta(n^2)$

Worst Case Partitioning

- The running time of quicksort depends on whether the partitioning is **balanced** or not.
  - $\Theta(n)$ time to partition an array of $n$ elements
  - Let $T(n)$ be the time needed to sort $n$ elements
  - $T(0) = T(1) = c$, where $c$ is a constant
  - When $n > 1$,
    - $T(n) = T(|left|) + T(|right|) + \Theta(n)$
  - $T(n)$ is maximum (worst-case) when either $|left| = 0$ or $|right| = 0$ following each partitioning
**Worst Case Partitioning**

- Worst-Case Performance (unbalanced):
  - $T(n) = T(1) + T(n-1) + \Theta(n)$
  - partitioning takes $\Theta(n)$
  - $= (2 + 3 + 4 + \ldots + n-1 + n) + n = \sum_{k=2}^{n} \Theta(k) + n = \Theta(\sum_{k=2}^{n} k) + n = \Theta(n^2)$

- This occurs when
  - the input is completely sorted
  - or when
    - the pivot is always the smallest (largest) element

---

**Best Case Partitioning**

- When the partitioning procedure produces two regions of size $\frac{n}{2}$, we get a balanced partition with best case performance:
  - $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

- Average complexity is also $\Theta(n \log n)$

---

*Figure 8.2* A recursion tree for Quicksort in which the **PARTITION** procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$.

---

*Figure 8.3* A recursion tree for Quicksort in which **PARTITION** always balances the two sides of the partition equally (the best case). The resulting running time is $\Theta(n \log n)$. 

---
**Average Case**

- Assuming random input, average-case running time is much closer to $\Theta(n \lg n)$ than $\Theta(n^2)$

- First, a more intuitive explanation/example:
  - Suppose that `partition()` always produces a 9-to-1 proportional split. This looks quite unbalanced!
  - The recurrence is thus:
    \[ T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n) \]
  - **How deep will the recursion go?**

---

**Average Case**

- Every level of the tree has cost $cn$, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most $cn$.
- The recursion terminates at depth $\log_{10}/9 n= \Theta(n \lg n)$.
- The total cost of quicksort is therefore $O(n \lg n)$.
- Intuitively, a real-life run of quicksort will produce a mix of “bad” and “good” splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case $n/2:n/2$ and worst-case $(n-1):1$
Intuition for the Average Case

- Suppose, we alternate lucky and unlucky cases to get an average behavior
  
  \[
  L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}
  \]
  
  \[
  U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}
  \]

  we consequently get
  
  \[
  L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n)
  \]
  
  which results in
  
  \[
  \Theta(n \log n)
  \]

The combination of good and bad splits would result in

\[ T(n) = \Theta(n \log n) \]

but with slightly larger constant hidden by the \( \Theta \)-notation.

Randomized Quicksort

An algorithm is randomized if its behavior is determined not only by the input but also by values produced by a random-number generator.

- This ensures that the pivot element is equally likely to be any of input elements.
- We can sometimes add randomization to an algorithm in order to obtain good average-case performance over all inputs.

Randomized Quicksort

Randomized-Partition(\( A, p, r \))
1. \( i \leftarrow \text{Random}(p, r) \)
2. exchange \( A[i] \leftrightarrow A[r] \)
3. return Partition(\( A, p, r \))

Randomized-Quicksort(\( A, p, r \))
1. if \( p < r \)
2. then \( q \leftarrow \text{Randomized-Partition}(A, p, r) \)
3. Randomized-Quicksort(\( A, p, q-1 \))
4. Randomized-Quicksort(\( A, q+1, r \))

Summary: Quicksort

- In worst-case, efficiency is \( \Theta(n^2) \)
  - But easy to avoid the worst-case
- On average, efficiency is \( \Theta(n \log n) \)
- Better space-complexity than mergesort.
- In practice, runs fast and widely used
  - Many ways to tune its performance
  - Can be combined effectively
- Various strategies for Partition