VE281

Data Structures and Algorithms

Merge Sort; Quick Sort

Outline

- Merge Sort

- Quick Sort

Review: Merge Sort

```
void mergesort(int *a, int left, int
  right) {
    if (left >= right) return;
    int mid = (left+right)/2;
    mergesort(a, left, mid);
    mergesort(a, mid+1, right);
    merge(a, left, mid, right);
}
```

$T(N/2)$

$T(N/2)$

$O(N)$

- Let $T(N)$ be the time required to merge sort N elements.

Recursive relation: $T(N) = 2T(N/2) + O(N)$

How to solve the recurrence?

Use master method

Solve Recurrence: Master Method

- Recurrence: $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$
  - Base case: $T(n) \le constant$ for all sufficiently small n.
  - $a =$ number of recursive calls (integer ≥ 1)
  - $b =$ input size shrinkage factor (integer > 1)
  - $O(n^d)$: the runtime of merging solutions. d is real value ≥ 0.
  - a, b, d are independent of n.
- <u>Claim</u>:

base doesn't matter

$$T(n) = \begin{cases} O(n^d \log n) & if\ a = b^d \\ O(n^d) & if\ a < b^d \\ O(n^{\log_b a}) & if\ a > b^d \end{cases}$$

base matters!

Example of Merge Sort

Recurrence: $$T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$$

Claim: $$T(n) = \begin{cases} O\left(n^d \log n\right) & if\ a = b^d \\ O\left(n^d\right) & if\ a < b^d \\ O\left(n^{\log_b a}\right) & if\ a > b^d \end{cases}$$

- $a = 2, b = 2, d = 1 \quad \Rightarrow b^d = a$
- $T(n) = O(n \log n)$

Another Example: Binary Search

Recurrence: $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

Claim: $T(n) = \begin{cases} O\left(n^d \log n\right) & if\ a = b^d \\ O\left(n^d\right) & if\ a < b^d \\ O\left(n^{\log_b a}\right) & if\ a > b^d \end{cases}$

- Exercise: What is a, b, d?

Merge Sort

Characteristics

- Not in-place
  - For efficient merging two sorted arrays, we need an auxiliary $O(N)$ space.
  - Recursion needs up to $O(\log N)$ stack space.

- Stable if **`merge()`** **maintains** the relative order of equal keys.

Divide-and-Conquer Approach

- Merge sort uses the **divide-and-conquer** approach.

- Recursively **breaking** down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.
  - For merge sort, split an array into two and sort them respectively.

- The solutions to the sub-problems are then **combined** to give a solution to the original problem.
  - For merge sort, merge two sorted arrays.

Outline

- Merge Sort
- Quick Sort

Quick Sort

Algorithm

Another divide-and-conquer approach to sort

- Choose an array element as **pivot**.
- Put all elements < pivot to the left of pivot.
- Put all elements ≥ pivot to the right of pivot.
- Move pivot to its correct place on the array.
- Sort left and right subarrays recursively (not including pivot).

$\left.\right\}$ `partition()`

```
void quicksort(int *a, int left,
  int right) {
    int pivotat; // index of the pivot
    if(left >= right) return;
    pivotat = partition(a, left, right);
    quicksort(a, left, pivotat-1);
    quicksort(a, pivotat+1, right);
}
```

Choice of Pivot

- If your input is random, you can choose the **first** element.
  - But this is very bad for presorted input.

- A better strategy: **randomly** pick an element from the array as pivot.
  - **Claim**: **for any input**, the average running time is $O(n \log n)$.
    - **Note**: average is over random choice of pivots made by the algorithm, **not** on the input.

Partitioning the Array

- Once pivot is chosen, swap pivot to the beginning of the array.
- When another array B is available, scan original array A from left to right.
  - Put elements < pivot at the left end of B.
  - Put elements ≥ pivot at the right end of B.
  - The pivot is put at the remaining position of B.
  - Copy B back to A.

A	6	2	8	5	11	10	4	1	9	7	3
B	2	5	4	1	3	6	7	9	10	11	8

In-Place Partitioning the Array

1. Once pivot is chosen, swap pivot to the beginning of the array.
2. Start counters **`i=1`** and **`j=N-1`**.
3. Increment **`i`** until we find element **`A[i]>=pivot`**.
   - **`A[i]`** is the leftmost item $\geq$ pivot.
4. Decrement **`j`** until we find element **`A[j]<pivot`**.
   - **`A[j]`** is the rightmost item $<$ pivot.
5. If **`i<j`**, swap **`A[i]`** with **`A[j]`**. Go back to step 3.
6. Otherwise, swap the first element (pivot) with **`A[j]`**.

In-Place Partitioning the Array

Example

i　　　j

A	6	2	8	5	11	10	4	1	9	7	3
A	6	2	3	5	11	10	4	1	9	7	8
A	6	2	3	5	1	10	4	11	9	7	8
A	6	2	3	5	1	4	10	11	9	7	8

- Now, **`j < i`**, swap the first element (pivot) with **`A[j]`**.

A	4	2	3	5	1	6	10	11	9	7	8

In-Place Partitioning the Array

Time Complexity

1. Once pivot is chosen, swap pivot to the beginning of the array.
2. Start counters **`i=1`** and **`j=N-1`**.
3. Increment **`i`** until we find element **`A[i]>=pivot`**.
4. Decrement **`j`** until we find element **`A[j]<pivot`**.
5. If **`i<j`**, swap **`A[i]`** with **`A[j]`**. Go back to step 3.
6. Otherwise, swap the first element (pivot) with **`A[j]`**.

- Scan the entire array no more than twice.
- Time complexity is $O(N)$, where N is the size of the array.

Quick Sort

Time Complexity

```
void quicksort(int *a, int left,
   int right) {
      int pivotat; // index of the pivot
      if(left >= right) return;
      pivotat = partition(a, left, right);
      quicksort(a, left, pivotat-1);
      quicksort(a, pivotat+1, right);
}
```

$O(N)$

$T(LeftSz)$

$T(RightSz)$

- Let $T(N)$ be the time needed to sort N elements.
  - $T(0) = c$, where c is a constant.
- Recursive relation:

$$T(N) = T(LeftSz) + T(RightSz) + O(N)$$

  - $LeftSz + RightSz = N - 1$

Quick Sort

Worst Case Time Complexity

- Recursive relation:

$$T(N) = T(LeftSz) + T(RightSz) + O(N)$$

- Worst case happens when each time the pivot is the smallest item or the largest item
  - $T(N) = T(N-1) + T(0) + O(N)$

$$\begin{aligned}
&\leq T(N-1) + T(0) + dN \\
&\leq T(N-2) + 2T(0) + d(N-1) + dN \\
&\quad \dots \\
&\leq T(0) + NT(0) + d + 2d + \cdots + d(N-1) + dN \\
&= O(N^2)
\end{aligned}$$

Quick Sort
Best Case Time Complexity

- Recursive realtaion:

$$T(N) = T(LeftSz) + T(RightSz) + O(N)$$

- Best case happens when each time the pivot divides the array into two equal-sized ones.
  - $T(N) = T((N-1)/2) + T((N-1)/2) + O(N)$
  - The recursive relation is similar to that of merge sort.
  - $T(N) = O(N \log N)$

Quick Sort
Average Case Time Complexity

- Average case time complexity of quick sort can be proved to be $O(N \log N)$.
  - Assume **randomly** pick an element from the array as pivot.
  - **Note**: average is over random choice of pivots made by the algorithm, **not** on the input.
  - The claim holds for any input.

Proof of Average Case Time Complexity

- Fix input array A of length N
- Sample space Ω: all possible pivot sequences that quick sort may choose
- Given random choice $\sigma \in \Omega$, define $C(\sigma)=$ total number of comparisons made by quicksort
  - $C(\sigma)$ is a random variable
- Lemma: running time of quicksort is dominated by # of comparisons
  - I.e., there exists a constant c so that for all $\sigma \in \Omega$,
$$RunTime(\sigma) \leq c \cdot C(\sigma)$$
- Remaining goal: $E[C] = O(N \log N)$

Proof of Average Case Time Complexity

- Define $z_i = i$-th smallest element of A

3	6	5	2
z_2	z_4	z_3	z_1

- For each $\sigma \in \Omega$, indices $i < j$,
  $X_{ij}(\sigma)$= # of times z_i, z_j get compared in quick sort with pivot sequence σ
- **Question**: what is the possible value of $X_{ij}(\sigma)$?
  - 0 or 1
  - **Reason**: two elements are compared only when one is the pivot. After that, they will not be compared any more

Proof of Average Case Time Complexity

- Important relation:

$$C(\sigma) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} X_{ij}(\sigma)$$

- By linearity of expectation:

$$E[C(\sigma)] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} E[X_{ij}(\sigma)]$$

0-1 random variable

- $E\left[X_{ij}(\sigma)\right] = \Pr(X_{ij} = 1)$
- Thus, $E[C(\sigma)] =$
  $\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \Pr(z_i, z_j \; get \; compared)$

Proof of Average Case Time Complexity

- Key claim: for all $i < j$,

$$\Pr(z_i, z_j \; get \; compared) = \frac{2}{j - i + 1}$$

- Proof of the key claim:
  - Fix z_i, z_j, consider the sequence $z_i, z_{i+1}, \dots, z_{j-1}, z_j$
  - As long as none of these are chosen as a pivot, all are passed to the same recursive call
  - Consider the first among $z_i, \dots, z_j$ that gets chosen as a pivot.

1. If z_i or z_j gets chosen first, then z_i and z_j are compared
2. If one of $z_{i+1}, \dots, z_{j-1}$ gets chosen first, then z_i and z_j are never compared: they are put into different recursive calls

Proof of Average Case Time Complexity

- Key claim: for all $i < j$,

$$\Pr(z_i, z_j \ get\ compared) = \frac{2}{j - i + 1}$$

- Proof of the key claim:
  1. If z_i or z_j gets chosen first, then z_i and z_j are compared
  2. If one of $z_{i+1}, \ldots, z_{j-1}$ gets chosen first, then z_i and z_j are never compared
  - Since pivot sequence is chosen uniformly at random, each of $z_i, z_{i+1}, \ldots, z_{j-1}, z_j$ is equally likely to be the first
  - Thus, $\Pr(z_i, z_j \ get\ compared) = \frac{2}{j-i+1}$

2: # choices lead to case 1

j-i+1: total # of choices

Proof of Average Case Time Complexity

- What we have so far: $E[C] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{2}{j-i+1}$
- Our target: $E[C] = O(N \log N)$

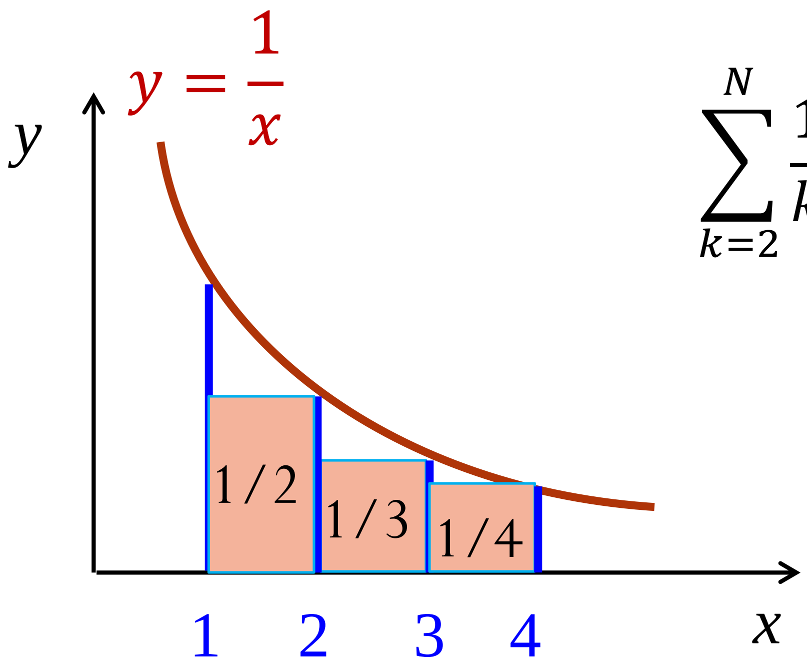
- Note: for each fixed $i \geq 1$,

$$\sum_{j=i+1}^{N} \frac{1}{j-i+1} \leq \sum_{j=i+1}^{N+i-1} \frac{1}{j-i+1} = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}$$

- Claim: $\sum_{k=2}^{N} \frac{1}{k} < \ln N$
- Once we prove the above claim, we get $E[C] < 2N \ln N$

Proof of the Claim

- Claim: $\sum_{k=2}^{N} \frac{1}{k} < \ln N$



$$\sum_{k=2}^{N} \frac{1}{k} < \int_{1}^{N} \frac{1}{x} dx = \ln N$$

Quick Sort

Average Case Time Complexity

- Average case time complexity of quick sort is $O(N \log N)$.
  - Assume **randomly** pick an element from the array as pivot.
  - **Note**: average is over random choice of pivots made by the algorithm, **not** on the input.
  - The claim holds for any input.

Quick Sort

Other Characteristics

- In-place?
  - In-place partitioning.
  - Worst case needs $O(N)$ stack space.
  - Average case needs $O(\log N)$ stack space.
    - "Weekly" in-place.

- Not stable.

Quick Sort

Summary

- Like merge sort, quick sort is a divide-and-conquer algorithm.
- Merge sort: easy division, complex combination.
- Quick sort: complex division (partition with pivot step), easy combination.
- Insertion sort is faster than quick sort for small arrays.
  - Terminate quick sort when array size is below a threshold. Do insertion sort on subarrays.