7: Quicksort

→ Worst-case running time : $\Theta(n^2)$
→ Expected running time : $\Theta(n \lg n)$
→ Constants hidden in $\Theta(n \lg n)$ are small
→ Sorts inplace.
→ Divide-Conquer strategy.

7.1 : Description of quicksort

Quicksort is based on the three-step process of divide-&-conquer

```
QUICKSORT(A,p,r)
① if p<r
② then q = PARTITION(A,p,r)
③      QUICKSORT(A,p,q-1)
④      QUICKSORT(A,q+1,r)
⑤ Initial call is QUICKSORT(A,1,n).
```

Partitioning

```
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
               exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

→ PARTITION always selects 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 regions, some of which may be empty:

Loop invariant:

① All entries in $A[p..i]$ are $\leq$ pivot

② All entries in $A[i+1..j-1]$ are $>$ pivot

③ $A[r]$ = pivot.

Correctness: Use the loop invariant to prove correctness of PARTITION:

Initialization: Before the loop starts, all the conditions of the loop invariant are satisfied, because r is the pivot & the subarrays $A[p..i]$ & $A[i+1..j-1]$ are empty.

Maintenance: While the loop is running, if $A[j] \leq$ pivot, then $A[j]$ & $A[i+1]$ are swapped & then i & j are incremented. If $A[j] >$ pivot, then increment only j.

Termination: When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases: $A[p..i] \leq$ pivot & $A[i+1..r-1] >$ pivot & $A[r]$ = pivot.

Time for partitioning: $\Theta(n)$ to partition an n-element subarray.

7.2 Performance of quicksort:

The running time of quicksort depends on the partitioning of the subarrays:

→ If the subarrays are balanced, then quicksort can run as fast as mergesort.

→ If they are unbalanced, then quicksort can run as slowly as insertion sort.

* Worst-Case

→ Occurs when the subarrays are completely unbalanced.

→ Have 0 elements in one subarray & $n-1$ elements in the other subarray.

→ Get the recurrence

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

→ Same running time as insertion sort.

→ In fact, the worst-case running time occurs when quicksort takes a sorted array as I/P. but insertion sort runs in $O(n)$ time in this case.

* Best-Case:

→ Occurs when the subarrays are completely balanced every time.

→ Each subarray has $\leq n/2$ elements.

→ Get the recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

* Balanced partitioning

→ Quicksort's average running time is much closer to the best case than to the worst case.

→ Imagine that PARTITION always produces a 9-to-1 split.

→ Get the recurrence

$$T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$$
$$= O(n \lg n)$$

→ Intuition : look at the recursion tree.

7.3:

Randomized version of quicksort

→ We have assumed that all I/P permutations are equally likely.

→ This is not always true.

→ To correct this, we added randomization to quicksort.

→ We could randomly permute the I/P array.

→ Instead, we use random sampling, on picking one element at random.

→ Don't always use A[r] as the pivot, Instead, randomly pick an element from the subarray that is being sorted.

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

Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

```
RANDOMIZED-QUICKSORT(A,p,r)
if p<r
  then q = RANDOMIZED-PARTITON(A,p,r)
       RANDOMIZED-QUICKSORT(A,p,q-1)
       "        "           (A,q+1,r).
```

Randomization of quicksort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array cause worst-case behavior in non-randomized QUICKSORT, but not in RANDOMIZED-QUICKSORT.

Exercise:

7.1-1 : Using figure 7.1 as a model, illustrate the operation of PARTITION on the array A = <13,19,9,5,12,8,7,4,21,2,6,11>.

A = < 13,19,9,5,12,8,7,4,21,2,6,11 > (A[r])

< 13,19,9,5,12,8,7,4,21,2,6,11 >

< 13,19,9,5,12,8,7,4,21,2,6,11 >

∵ A[j] ≤ A[r], i = i+1, A[i] ↔ A[j]

< 9,19,13,5,12,8,7,4,21,2,6,11 >

< 9,5,13,19,12,8,7,4,21,2,6,11 >

< 9,5,8,19,12,13,7,4,21,2,6,11 >

< 9,5,8,7,12,13,19,4,21,2,6,11 >

< 9,5,8,7,4,13,19,12,21,2,6,11 >

< 9,5,8,7,4,2,19,12,21,13,6,11 >

< 9,5,8,7,4,2,6,12,21,13,19,11 >

A[i+1] ↔ A[r]

< 9,5,8,7,4,2,6,11,21,13,19,12 >

7.1-2: What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value? Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the array $A[p..r]$ have the same value.

It returns r.

We can modify PARTITION by counting the number of comparisons in which $A[j] \leq A[r]$ & then subtracting that number from the pivot index.

7.1-3: Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

There is a for statement whose body executes $r - 1 - p = \Theta(n)$ times. In the worst-case every time the body of the if is executed, but it takes constant time & so does the code outside of the loop. Thus the running time is $\Theta(n)$.

7.1-4: How would you modify QUICKSORT to sort into nonincreasing order?

We only need to flip the condition on the first.

7.2: Performance of quicksort

7.2-1: Use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of section 7.2.

We represent $\Theta(n)$ as $c_2 n$ & we guess that $T(n) \leq c_1 n^2$

$$\begin{aligned} T(n) &= T(n-1) + c_2 n \\ &\leq c_1 (n-1)^2 + c_2 n \\ &= c_1 n^2 - 2c_1 n + c_1 + c_2 n \qquad (2c_1 > c_2, \\ &\leq c_1 n^2. \qquad n \geq c_1/(2c_1 - c_2) \end{aligned}$$

7.2-2: What is the running time of QUICKSORT when all elements of the array A have the same value?

It is $\Theta(n^2)$, since one of the partitions is always empty (exercise 7.1-2)

7.2-3: Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements & is sorted in decreasing order.

If the array is already sorted in decreasing order, then, the pivot element is less than all the other elements. The partition step takes $\Theta(n)$ time, & then leaves you with a subproblem of size $n-1$ & a subproblem of size 0. This gives us the recurrences considered in 7.2-1. which we showed has a solution that is $\Theta(n^2)$.

7.2-4: Bank often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check numbers. People usually write checks in orders by check number, & & merchants usually cash them with reasonable dispatch. The problem of converting time-of-transactions ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure

INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

The more sorted the array is, the less work insertion sort will do. Namely, INSERTION-SORT is $\Theta(n+d)$, where d is the number of inversions in the array. In the above example, the number of inversions tends to be small so insertion sort will be close to linear.

On the other hand, if PARTITION does pick pivot that does not participate in an inversion, it will produce an empty partition. Since there is a small number of inversions, QUICKSORT is very likely to produce empty partitions.

7.2-5: Suppose that the splits at every level of quicksort are in proportion $1-\alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ & the maximum depth is approximately $-\lg n / \lg(1-\alpha)$.

7.3: A randomized version of quicksort

7.3-1: Why do we analyze the expected running time of a randomized algo. & not its worst-case running time?

We analyze the expected run time because it represents the more typical time cost. Also, we are doing the expected run time over the possible randomness used during computation because it can't be produced adversarially, unlike when doing expected run time over all possible I/Ps to the algorithms.

7.3-2: When RANDOMIZED-QUICKSORT runs, how many calls are made to the random number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

In the worst case, the number of calls to RANDOM is

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

As for the best case,

$$T(n) = 2T(n/2) + 1 = \Theta(n)$$

This is not too surprising, because each third element (at least) gets picked as pivot.

7.4 Analysis of quicksort

7.4-1: Show that in the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n),$$

$$T(n) = \Omega(n^2).$$

We guess $T(n) \ge cn^2 - 2n$,

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

$$\ge \max_{0 \le q \le n-1} (cq^2 - 2q + c(n-q-1)^2) - 2n - 2q - 1) + \Theta(n)$$

$$\ge c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2 - (2n + 4q - 1)/c) + \Theta(n)$$

$$\ge cn^2 - c(2n-1) + \Theta(n)$$

$$\ge cn^2 - 2cn + 2c$$

$$\ge cn^2 - 2n. \qquad [\because c \le 1]$$

7.4-2: Show that quicksort's best-case running time is $\Omega(n \lg n)$.

We'll use the substitution method to show that the best case running time is $\Omega(n \lg n)$. Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have.

CH22: HASHING INTRO
DATE:

$$T(n) = \min_{1 \le q \le n-1} \left(T(q) + T(n-q-1) \right) + \Theta(n)$$

Suppose that $T(n) \ge c(n \lg n + 2n)$ for some constant c. Substituting this gets guess into the recurrence gives.

$$T(n) \ge \min_{1 \le q \le n-1} \left(cq \lg q + 2cq + c(n-q-1)\lg(n-q-1) + 2c(n-q-1) \right) + \Theta(n).$$

$$= (cn/2)\lg(n/2) + cn + c(n/2-1)\lg(n/2-1) + cn - 2c + \Theta(n)$$

$$\ge (cn/2)\lg n - cn/2 + c(n/2-1)(\lg n - 2) + 2cn - 2c\Theta(n)$$

$$= (cn/2)\lg n - cn/2 + (cn/2)\lg n - cn - c\lg n + 2c + 2cn - 2c\Theta(n)$$

$$= cn\lg n + cn/2 - c\lg n + 2c - 2c\Theta(n).$$

Taking a derivative with respect to q shows that the minimum is obtained when $q = n/2$. Taking c large enough to dominate the $-\lg n + 2 - 2c + \Theta(n)$ term makes this greater than $cn\lg n$, proving the bound.

7.4-3: Show that the expression $q^2 + (n-q-1)^2$ achieves a maximum over $q = 0, 1, \ldots, n-1$ when $q = 0$ or $q = n-1$.

$$f(q) = q^2 + (n-q-1)^2$$

$$f'(q) = 2q - 2(n-q-1) = 4q - 2n + 2$$

$$f''(q) = 4.$$

$f'(q) = 0$ when $q = \frac{1}{2}n - \frac{1}{2}$. $f'(q)$ is also continious. $\forall q : f''(q) > 0$, which means that $f'(q)$ is negative left of $f'(q) = 0$ & positive right of it, which means that this is a local minima. In this case, $f(q)$ is decreasing in the beginning of the interval & increasing in the end, which means that those two points are the only candidates for a maximum in the interval.

$$f(0) = (n-1)^2$$

$$f(n-1) = (n-1)^2 + 0^2$$

7.4-4: Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

We use the same reasoning for the expected number of comparisions, we just take in a different direction.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \qquad (k \ge 1)$$

$$E[X] \geq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{2k}$$

$$\geq \sum_{i=1}^{n-1} \Omega(\lg n)$$

$$\boxed{E[X] = \Omega(n \lg n)}$$

Using the master method, we get the solution $\Theta(n \lg n)$.

7.4-5: We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its I/p is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should we pick k, both in theory & practice?

In the quicksort part of the proposed algorithm, the recursion stops at level $\lg(n/k)$, which makes the expected running time $O(n \lg(n/k))$. However, this leaves n/k non-sorted, non-intersecting subarrays of (maximum) length k.

Because the nature of the insertion sort algorithm, it will first sort fully one such subarray before considering the next one. Thus, it has the same complexity as sorting each of those arrays, that is $\frac{n}{k} O(k^2) = O(nk)$.

In theory, if we ignore the constant factors, we need to solve

$$n \lg n \geq nk + n \lg n/k$$

$$\Rightarrow \lg n \geq k + \lg n - \lg k$$

$$\Rightarrow \lg k \geq k.$$

Which is not possible.

If we add the constant factors, we get,

$$c_q n \lg n \geq c_i nk + c_q n \lg(n/k)$$

$$\Rightarrow c_q \lg n \geq c_i k + c_q \lg n - c_q \lg k$$

$$\Rightarrow \lg k \geq \frac{c_i}{c_q} k.$$

which indicates that there might be a good candidate. Furthermore, the lower-order terms should be taken into consideration too.

In practice, k should be chosen by experiments.

WEEK 2: HASHING INTRO &
DATE:

* 7.4-6: Consider modifying the PARTITION procedure by randomly picking three elements from array A & partitioning about their median (the middle value of the three elements). Approximate the probability of getting at worst an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

First, for simplicity's sake, Let's assume that we can pick the same element twice. Let's also assume that $0 < \alpha \leq 1/2$.

In order to get such a split, two out of three elements need to be in the smallest αn elements. The probability of having one is $\alpha n/n = \alpha$. The probability of having exactly two is $\alpha^2 - \alpha^3$. There are three ways in which two elements can be in the smallest αn & one way in which all three can be in the smallest αn so the probability of getting such a median is $3\alpha^2 - 2\alpha^3$. We will get the same split if the median is in the largest αn. Since the two events are mutually exclusive, the probability is

$$P_{r}(\text{OK split}) = 6\alpha^2 - 4\alpha^3 = 2\alpha^2(3-2\alpha)$$

8: Sorting in Linear Time

Theorem: Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

* Proof: → $l \geq n!$

→ By lemma: $n! \leq l \leq 2^h$ or $2^h \geq n!$

Take log both side $h \geq \lg(n!)$

Use Stirling approximation: $n! > (n/e)^n$

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \end{aligned}$$

* Proof by induction on h.

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$

Inductive step: Assume true for height $= h-1$.

Extend tree of height $h-1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

# of leaves for height $h = 2 \cdot$ (# of leaves for height $h-1$)