Quicksort

Fastest Sorting Algorithm on Average, How To Prove That

Quicksort: Different Kind Of Divide & Conquer

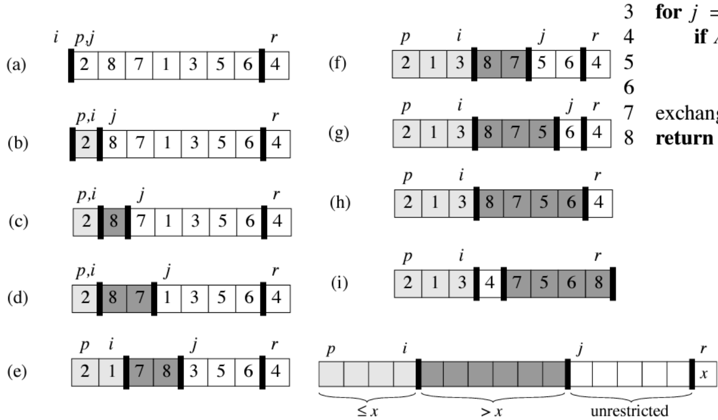
- So far, "divide" was straightforward, and "conquer" was involved.
- In sorting, can we make "conquer" part easy (almost nothing), by doing more on "divide" part?
  - CLRS 7.1 "Divide": **<u>Partition</u>** (rearrange) the array $A[p..r]$ into two (either one of the two may be empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that:
    - $A[i] \le A[q]$ for any $p \le i < q$, and
    - $A[j] > A[q]$ for any $q < j \le r$.
  - CLRS 7.1 "Conquer": Then conquering becomes straightforward:
    - Sort $A[p..q-1]$ recursively
    - Sort $A[q+1..r]$ recursively

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

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

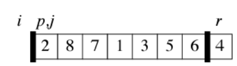
PARTITION(A, p, r) Illustration

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



PARTITION() Can Be Recursive As Well

- Maybe more overhead, but maybe easier to understand



- If $A[p] \le A[r]$, return PARTITION$(A, p+1, r)$
- Otherwise, 3-way swap between $A[p]$, $A[r]$, and $A[r-1]$
  - $A[p]$ to $A[r]$, $A[r]$ to $A[r-1]$, $A[r-1]$ to $A[p]$, then return PARTITION$(A, p, r-1)$
  - Definitely more swaps (so more overhead), but still correct (and same asymptotic notation) with easier derivation of the recurrence relation
    - $T(n) = T(n-1) + \Theta(1) \rightarrow T(n) = \Theta(n)$
- Base case: If $p = r$, return r.

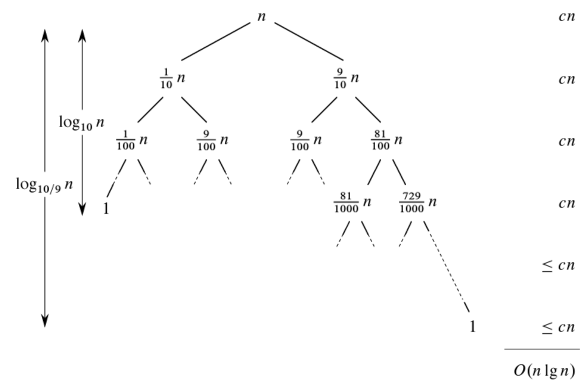
Quicksort Analysis

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

- From the QUICKSORT() pseudocode, $T_{qsort}(n) =$
- It's obvious that $T_{partition}(n) = \Theta(n)$.
- So, $T(n) = T(n_1) + T(n_2) + \Theta(n)$ where $n_1 + n_2 + 1 = n$
- Worst case: $n_1 = 0$ or $n_2 = 0$ all the time (bad split/partition) →
    - $T(n) = T(n-1) + \Theta(n)$ → $T(n) = \Theta(n^2)$
        - When would this happen? What's the insertion/bubble sort performance in that case?
- Best case: $n_1 \cong n_2$ as much as possible (even split) →
    - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ → $T(n) = \Theta(n \lg n)$

Balanced Splits, Even Skewed (CLRS Fig. 7.4)

- 9-to-1 splits all the time
- In fact, doesn't matter what x & y in x-to-y splits, as long as x & y are fixed.



Randomized Quicksort (CLRS Section 7.3)

- To avoid worst case as much as possible,
    - Pick the pivot from a random index, not from a fixed one at the end.
    - Still rely on the original PARTITION() after swapping the randomly picked pivot with the original fixed pivot.

```
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)
```

Formal Proofs of RANDOMIZED-QUICKSORT Time Complexities (CLRS Section 7.4)

- Lots of algebraic derivations. We won't focus on those.
- Also random probabilistic analysis and derivations for randomized case. We won't focus on those either.
- Just read through CLRS Section 7.4 and see how they go.

Brief Summary

- Worst-case: $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$
  - We can show $T(n) \le cn^2$ for some c and large enough n, showing $T(n) = O(n^2)$ (This is done in textbook)
  - Can also show $T(n) \ge cn^2$ for some c and large enough n, showing $T(n) = \Omega(n^2)$ (Exercise 7.4-1)
  - Therefore, worst-case $T(n) = \Theta(n^2)$.
- Average-case (expected running time) of RANDOMIZED-QUICKSORT()
  - Probabilities of possible cases, number of comparisons becoming random variable, derive the expected average of the random variable.
  - $E[X] = \cdots = O(n \log n)$

Median Finding Algorithm

No Need To Sort Entire Array

Medians and Order Statistics

- Given a set A of n elements,
  - Minimum (first in the ordered sequence), maximum (last), median (mid)
    - If n is even, there could be 2 medians. For simplicity, we mean the lower median.
- General i-th order statistic: The i-th smallest element of A
  - Minimum: A's 1st order statistic, maximum: A's n-th order statistic
  - Median: A's $\lfloor (n+1)/2 \rfloor$-th order statistic
- Algorithm to find the i-th order statistic of A for any given A and i
  - Simple (Naïve): Sort A, return $A[i]$: $O(n \lg n)$
  - Do we really need to sort the entire array? Aren't we doing more than necessary?

Average Linear Time Selection Algorithm

- CLRS 9.2 RANDOMIZED-SELECT() (pp. 216)

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k          // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Time Complexity Analysis of RAND-SEL

- Worst case: $\Theta(n^2)$, just like quicksort
- Average (expected) case
  - Requires random variable analysis, just like in quicksort
    - Assume probabilities, find expected running time, show it's at most $O(n)$
  - Details in CLRS 9.2
    - We don't need to do this all the time.
    - I'd say intuition is more important than formal proof.
      - Think about balanced splits-case (e.g., 2:3), and derive running time, confirm it's $O(n)$.
- Can we achieve $O(n)$ in worst case as well?
  - Surprisingly, yes. Study CLRS 9.3 (left as optional).
    - Time complexity analysis of SELECT() is more interesting and involved.

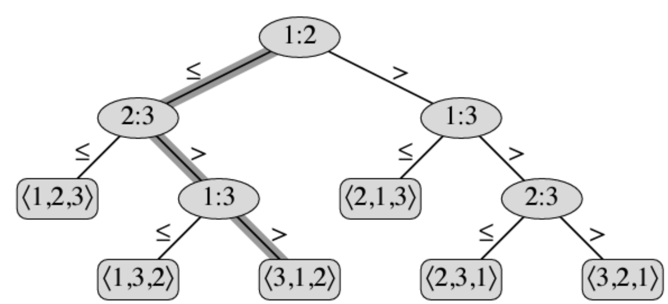
Lower Bounds For Sorting

How Fast Can We Do Sorting By Comparing

Sorting based on Comparisons

- All sorting algorithms we learned so far are based on:
  - Repeatedly *comparing* two elements of the given array.
- We've seen as good as $\Theta(n \lg n)$ comparison sorting algorithms.
  - Merge sort (all cases), quicksort (average/expected cases), heapsort (will be covered later, all cases)
- Is there any better comparison sorting algorithms?
  - Surprisingly (or not) no.
  - It's proven by analyzing any comparison-based sorting algorithm:
    - A sequence of comparisons, determining the final total order.
    - Starting from one pair, its comparison determining next comparison, …
      - We get a so-called decision tree.

Decision-Tree Model



Lower Bound For Worst Case

- "Takes at least this long in the worst case"
  - The height of the decision tree!
- There are $n!$ permutations for any given input array of size n.
  - Every permutation must show up as a leaf in the decision tree:
    - $n! \le L$ (L is the number of leaves in the decision tree)
  - For a binary tree of height h, there are at most 2^h leaves:
    - $L \le 2^h$
- Therefore, we get $n! \le 2^h$.
- Solving for h, we get:
  - $h \ge \log_2(n!) = \log_2 n + \log_2(n-1) + \cdots = \Omega(n \lg n)$ (eq. (3.19) in pp. 58)

- "Takes at least this long in the worst case"
  - The height of the decision tree!
- There are $n!$ permutations for any given input array of size n.
  - Every permutation must show up as a leaf in the decision tree:
    - $n! \le L$ (L is the number of leaves in the decision tree)
  - For a binary tree of height h, there are at most 2^h leaves:
    - $L \le 2^h$
- Therefore, we get $n! \le 2^h$.
- Solving for h, we get:
  - $h \ge \log_2(n!) = \log_2 n + \log_2(n-1) + \cdots = \Omega(n \lg n)$ (eq. (3.19) in pp. 58)

Sorting In Linear Time

Do We Always Have To Compare To Sort?

Counting Sort

- When there are a lot more elements than possible distinct values
  - E.g.: 1,0,2,0,0,1,1,2,0,1,2,0 ← Only 3 possible distinct values, but 12 elements
- Count the number of occurrences of each value, create the "counts" array:
- Then reproduce the sorted sequence out of the counts
- Experiment counting sort at http://visualgo.net/sorting

Example: 2, 5, 3, 0, 2, 3, 0, 3 (CLRS Fig. 8.2)

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C	1	2	4	5	7	8

	0	1	2	3	4	5
C	2	0	2	3	0	1

```
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
```

Counting Sort Time Complexity

- Initializing counts array: $\Theta(k)$ (k is the largest possible value)
- Counting/constructing part: $\Theta(n)$
- Therefore, $\Theta(k+n)$.
- If $k = O(n)$, then $\Theta(n)$.
  - The premise ($k = O(n)$) is important!
  - If k is arbitrarily large (e.g., a double value) and n is not that big (e.g., 100), you don't want to use this algorithm!
- CLRS pp. 195 COUNTING-SORT() pseudocode
  - More involved to meet the "stability" requirement
    - Important for radix sort.

Radix Sort

- Sort *discrete* values digit-by-digit repeatedly in d passes
  - However, start from least-significant digit, and move up! (Counterintuitive)
- Experiment radix sort at http://visualgo.net/sorting
- Example: 329, 457, 657, 839, 436, 720, 355
- Why does it work? How to prove? Use induction on # digits
  - "Stability" in digit-by-digit sorting is important!
- Time complexity: $\Theta(d(n+k))$. If d and k are constants, it's $\Theta(n)$.

Bucket Sort

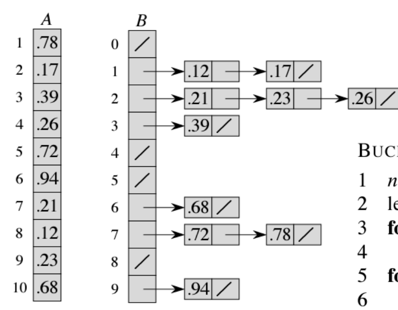
- Only good for input array when its values are *uniformly distributed* over the interval [min, max]
  - Divide the interval into n equal-sized subintervals, or "buckets"
  - Distribute the n input numbers into the buckets
    - Because of the "uniformly distributed" assumption, each bucket shouldn't contain too many elements
    - Thus, sorting elements in each bucket should be bound to a constant.
  - Final sorting is to collect elements from each bucket one-by-one after sorting elements in each bucket.
- Time complexity analysis: Another probability & random var. analysis
  - $\Theta(n)$, on average, again only under the *uniformly distributed* assumption

Bucket Sort Example and Code (CLRS Fig. 8.4)

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \end{aligned}$$

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n$$



BUCKET-SORT(A)

```
n = A.length
let B[0 . . n − 1] be a new array
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Balancing the Work

How we optimize some divide and conquer problems?

Search in a Sorted Array with Limited Resources

- What is the tallest floor from which I can drop an egg without breaking it? Say you have *n* floors and *k* eggs
- If there is only one egg, we must do sequential search from the bottom floor
- If we have many eggs, say *k* at least $\log_2 n$, we can use binary search.
- In between, we can divide the building in *a* parts and solve the problem recursively, dropping the egg from the top of each part.
- Then we have:

$$T(n,k) = a + T(n/a,k-1) \qquad \& \quad T(n,1) = n.$$

Search in a Sorted Array with Limited Resources

- Expanding we have:

$$T(n,k) = a + a + T(n/a^2,k-2)$$

$$\dots\dots\dots$$

$$T(n,k) = \underbrace{a + a + \dots\dots\dots\dots\dots + a}_{k-1} + T(n/a^{k-1},1)$$

$$T(n,k) = a + a + \dots\dots\dots\dots\dots + a + n/a^{k-1}$$

- Balancing to optimize: all eggs should do the same work: $a = n/a^{k-1}$
- Hence: $a = n^{1/k}$ and then $T(n,k) = k\, n^{1/k}$
- For example for $k=2$, we have $T(n,2) = O(\sqrt{n})$