Quicksort

Randomized Algorithms: Week 2
Summer HSSP 2023
Emily Liu

Sorting Algorithms

Task: Given a list A of **comparables**, arrange them in increasing order.

- Comparable: for all elements (a, b) in A, we can say exactly one of the following is true:
    - $a > b$
    - $a < b$
    - $a = b$

Question: What sorting algorithms do you know?

Selection Sort

```
SELECTION_SORT(A):
  for i = 1 ... N-1:
    min_idx = i
    for j = i+1 ... N:
      if A[j] < A[min_idx]:
        min_idx = j
      endif
    endfor
    swap A[min_idx], A[i]
  endfor
```

High level algorithm: At each step, find minimum element in unsorted portion of array, swap with current element.

Questions:

- What is the time complexity of selection sort?
- What is the space complexity of selection sort?

Insertion Sort

```
INSERTION_SORT(A):
  for j = 2 ... N:
    key = A[j]
    i = j - 1
    while (A[i] > key and i > 0):
      swap(A[i+1], A[i])
      i++
    endwhile
    A[i+1] = key
  endfor
```

High level algorithm: At each step, take first element of unsorted portion of the array, insert it into the right place in the sorted portion of the array.

Questions:

- What is the time complexity of insertion sort?
- What is the space complexity of insertion sort?

Merge Sort

```
MERGE_SORT(A):
  A_left = MERGE_SORT(A[0:N/2])
  A_right = MERGE_SORT(A[N/2:N])
  A = MERGE(A_left, A_right)
MERGE(A_left, A_right):
  l, r, count = 0; merged = empty array
  while count < N:
    merged = min(A_left[l], A_right[r])
    if A_left[l] < A_right[r]: l++, else: r++
    count++
  endwhile
```

High level algorithm: Recursively break down the array into halves and sort, then merge the sorted halves in linear time.

Questions:

- What is the time complexity of merge sort?
- What is the space complexity of merge sort?

Comparison of Sorting Algorithms

Algorithm	**Time Complexity**	**Space Complexity**
Selection Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$ average/worst case $O(n)$ best case	$O(1)$
Merge Sort	$O(n \log n)$	$O(n)$
In-place Merge Sort	$O(n^2 \log n)$	$O(1)$

Quick Sort

QUICK_SORT(A):

```
1. Select a pivot index, p.
2. Denote subarrays L (less), E (equal), G (greater)
3. Rearrange array such that:
     all elements in L are to the left of A[p],
     all elements in G are to the right of A[p]
4. Recursively sort L and G using QUICK_SORT.
```

Space complexity of Quicksort

Claim: We can perform quicksort in-place.

Procedure:

1. Select A[N-1] (last element) to be pivot.
2. Define counters L = 0 … N-2; R = N-2 … 0.
3. Advance L and R until A[L] > pivot, A[R] < pivot
4. Swap A[L], A[R]
5. Continue until L = R
6. Insert pivot into right place

Question: What is the time complexity of this procedure?

Pivot Selection

Quicksort algorithm:

1. **Select pivot**
2. Divide L, E, G
3. Recurse

How to select pivot?

Proposal: select first (A[0]) or last (A[N-1]) element.

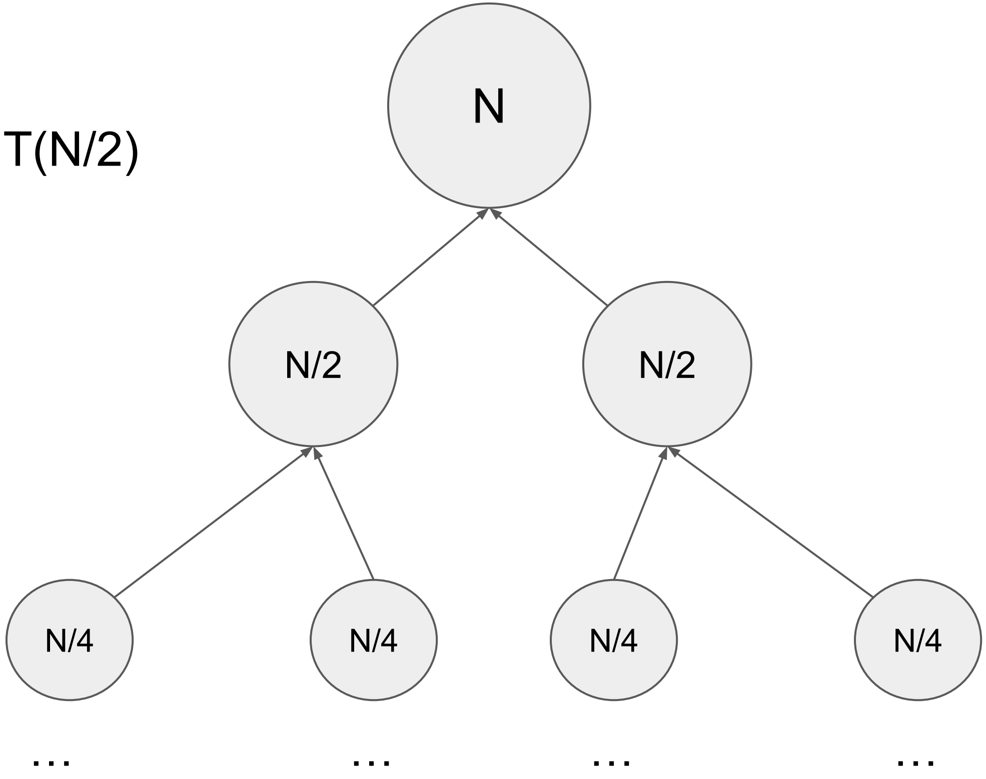
Assume: all elements distinct, any permutation of elements in A is equally likely

What is the **expected** runtime?

Runtime Analysis of Quicksort

Average case:

$$T(N) = O(N) + 2\ T(N/2)$$



Runtime Analysis of Quicksort

Worst case:

$T(N) = O(N) + O(1) + T(N-1)$

$T(N-1) = O(N-1) + O(1) + T(N-2)$

$T(N-2) = O(N-1) + O(1) + T(N-3)$

…

Worst case is **$O(n^2)$**, which is not very good!

Can use **randomization** to improve.

Randomized (“Paranoid”) Quicksort

TLDR: Naive pivot selection works well in expectation (and in practice!), but a poorly selected pivot can make quicksort very slow.

Want to know: Is there a good way to select a pivot that guarantees a good “split”, without compromising the runtime of the algorithm?

Claim: Randomly selecting the pivot is pretty good

Randomized ("Paranoid") Quicksort

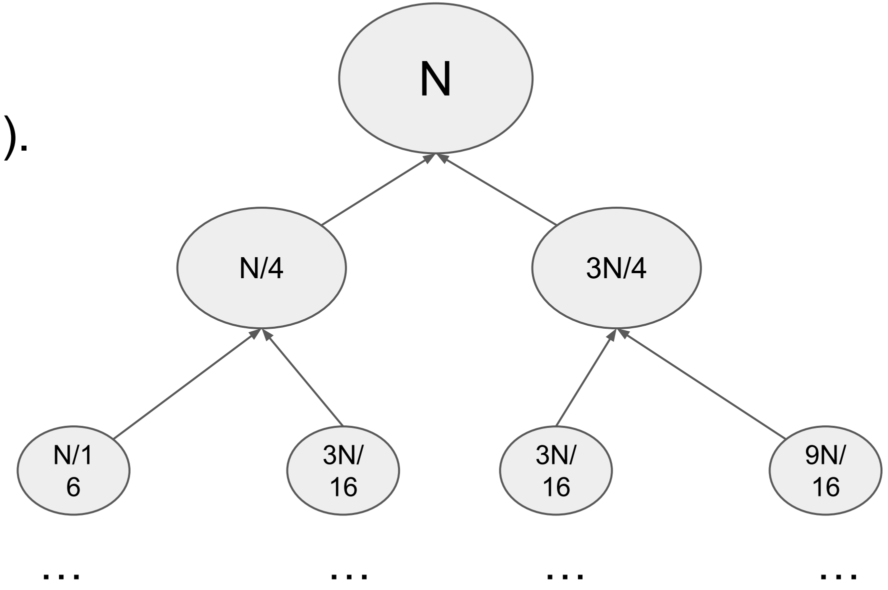
Suppose we can select a pivot in O(n) time such that

$$N/4 \le |L|, |G| \le 3N / 4.$$

Then,

$$T(n) \le O(n) + T(n/4) + T(3n/4).$$

Also evaluates to O(n log n).



Pivot Selection

Now, need to select pivot in O(n) time.

Question: If we randomly select from all indices in the array, what is the probability that the pivot we select is “good”?

bad pivots	good pivots	bad pivots
$\frac{n}{4}$	$\frac{n}{2}$	$\frac{n}{4}$

P(good) = ½ $\Rightarrow$ in expectation, repeat the process **twice**.

Conclusion

Quicksort:

- Expected time complexity: O(n log n)
- Space complexity: O(1)
- In practice:
  - Can assume O(n log n) even without checking if the pivot is “good”

Quicksort vs Merge sort:

- Quicksort: When space is more important
- Merge sort: Better on very large datasets, or when stability is important

Exercises

Warmup: Implement a not-in-place version of quicksort.

1. Implement in-place quicksort, using the last element of the array as a pivot.
2. Implement the “paranoid” randomized quicksort where you repeatedly select a pivot until it’s “good”, meaning the minimum size of L or G is at least N/4. Remember to keep the partitioning in-place!
    a. Hint: you can do this with very little modification to your code from part 1.
    b. Try experimenting with different “parameters” for pivot selection: eg, what if you set the minimum size as L/3? L/5?
3. Challenge: Come up with a quicksort algorithm that is guaranteed to run in O(n log n) time.
    a. Hint: https://brilliant.org/wiki/median-finding-algorithm/ may be of use.
    b. Don’t worry if your code for this runs slower than your code from part 2, or even from part 1. Remember that time complexities are *asymptotic* metrics, meaning that a good-looking time complexity may still have a long runtime due to high constant overheads.

Tip: Use datetime libraries to track the amount of time each algorithm takes to run.