Lecture 5
Randomized algorithms and QuickSort
Announcements

• HW2 is posted! Due Friday.

• Please send any OAE letters to Jessica Su (stysu@stanford.edu) by Friday.
Last time

- We saw a divide-and-conquer algorithm to solve the Select problem in time $O(n)$ in the worst-case.
- It all came down to picking the pivot...

![Graph](image_url)

We choose a pivot **randomly** and then a bad guy gets to decide what the array was.

We choose a pivot **cleverly** and then a bad guy gets to decide what the array was.

The bad guy gets to decide what the array was and then we choose a pivot **randomly**.
Randomized algorithms

- We make some random choices during the algorithm.
- We hope the algorithm works.
- We hope the algorithm is fast.

**Select** with a random pivot is a randomized algorithm.
- It always works.
- It is usually fast.
- It might be slow.
Today

• How do we analyze randomized algorithms?
• A few randomized algorithms for sorting.
  • BogoSort
  • QuickSort

• BogoSort is a pedagogical tool.
• QuickSort is important to know. (in contrast with BogoSort...)
How do we measure the runtime of a randomized algorithm?

**Scenario 1**
1. Bad guy picks the input.
2. You run your randomized algorithm.

**Scenario 2**
1. Bad guy picks the input.
2. Bad guy chooses the randomness (fixes the dice)

- In **Scenario 1**, the running time is a random variable.
  - It makes sense to talk about expected running time.
- In **Scenario 2**, the running time is not random.
  - We call this the worst-case running time of the randomized algorithm.
Today

• How do we analyze randomized algorithms?
• A few randomized algorithms for sorting.
  • BogoSort
  • QuickSort

• BogoSort is a pedagogical tool.
• QuickSort is important to know. (in contrast with BogoSort...)
• BogoSort(A):
  • While true:
    • Randomly permute A.
    • Check if A is sorted.
    • If A is sorted, return A.

• What is the expected running time?
  • You analyzed this in your pre-lecture exercise [also on board now]

• What is the worst-case running time?
  • [on board]
Today

• How do we analyze randomized algorithms?
• A few randomized algorithms for sorting.
  • BogoSort
  • QuickSort

• BogoSort is a pedagogical tool.
• QuickSort is important to know. (in contrast with BogoSort...)
a better randomized algorithm: **QuickSort**

- Runs in expected time $O(n \log(n))$.
- Worst-case runtime $O(n^2)$.
- In practice often more desirable.
  - (More later)
Quicksort

We want to sort this array.

First, pick a “pivot.” Do it at random.

Next, partition the array into “bigger than 5” or “less than 5”

L = array with things smaller than A[pivot]
R = array with things larger than A[pivot]

Recurse on L and R:
PseudoPseudoCode for what we just saw

- **QuickSort(A):**
  - **If** `len(A) <= 1:`
    - **return**
  - Pick some `x = A[i]` at random. Call this the **pivot**.
  - **PARTITION** the rest of `A` into:
    - `L` (less than `x`) and
    - `R` (greater than `x`)
  - Replace `A` with `[L, x, R]` (that is, rearrange `A` in this order)
  - `QuickSort(L)`
  - `QuickSort(R)`

Assume that all els of `A` are distinct. How would you change this if that’s not the case?

How would you do all this in-place? Without hurting the running time?
Running time?

• \( T(n) = T(|L|) + T(|R|) + O(n) \)

• In an ideal world... if the pivot splits the array exactly in half...
  \[
  T(n) = 2 \cdot T \left( \frac{n}{2} \right) + O(n)
  \]

• We’ve seen that a bunch: 
  \[
  T(n) = O(n \log(n)).
  \]
A tempting argument

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
- Therefore, the expected running time is $O(n \log(n))$.

This is not okay!!!
that’s not how expectations work.
[Discussion on board]
Instead

• We’ll have to think a little harder about how the algorithm works.

Goal for the rest of the class

• Get the same conclusion, correctly!
Example of recursive calls

7 6 3 5 1 2 4

Pick 5 as a pivot

3 1 2 4 5 7 6

Partition on either side of 5

Recurse on [3142] and pick 3 as a pivot.

Partition around 3.

1 2 3 4 5 6 7

Recurse on [76] and pick 6 as a pivot.

Partition on either side of 6

Recurse on [12] and pick 2 as a pivot.


Partition around 2.

1 2 3 4 5 6 7

Recurse on [7], it has size 1 so we’re done.

Recurse on [1] (done).

Recurse on [12] (done).
How long does this take to run?

- We will count the number of *comparisons* that the algorithm does.
  - This turns out to give us a good idea of the runtime. (Not obvious).
- How many times are any two items compared?

In the example before, everything was compared to 5 once in the first step....and never again.

But not everything was compared to 3. 5 was, and so were 1,2 and 4. But not 6 or 7.
Each pair of items is compared either 0 or 1 times. Which is it?

| 7 | 6 | 3 | 5 | 1 | 2 | 4 |

Let’s assume that the numbers in the array are actually the numbers 1,…,n

Of course this doesn’t have to be the case! It’s a good exercise to convince yourself that the analysis will still go through without this assumption. (Or see CLRS)

- Whether or not $a,b$ are compared is a random variable, that depends on the choice of pivots. Let’s say

$$X_{a,b} = \begin{cases} 
1 & \text{if } a \text{ and } b \text{ are ever compared} \\
0 & \text{if } a \text{ and } b \text{ are never compared}
\end{cases}$$

- In the previous example $X_{1,5} = 1$, because item 1 and item 5 were compared.
- But $X_{3,6} = 0$, because item 3 and item 6 were NOT compared.
- Both of these depended on our random choice of pivot!
Counting comparisons

• The number of comparisons total during the algorithm is

\[ \sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b} \]

• The expected number of comparisons is

\[ E \left[ \sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b} \right] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} E[ X_{a,b} ] \]

using linearity of expectations.
Counting comparisons

• So we just need to figure out $E[ X_{a,b} ]$

• $E[ X_{a,b} ] = P( X_{a,b} = 1 ) \cdot 1 + P( X_{a,b} = 0 ) \cdot 0 = P(X_{a,b} = 1)$
  
• (using definition of expectation)

• So we need to figure out

$P(X_{a,b} = 1) =$ the probability that $a$ and $b$ are ever compared.

Say that $a = 2$ and $b = 6$. What is the probability that 2 and 6 are ever compared?

This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.

If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.
Counting comparisons

\[ P(X_{a,b} = 1) \]

= probability \( a,b \) are ever compared

= probability that one of \( a,b \) are picked first out of all of the \( b - a + 1 \) numbers between them.

\[ = \frac{2}{b - a + 1} \]
All together now...

Expected number of comparisons

• \( E \left[ \sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b} \right] \)

This is the expected number of comparisons throughout the algorithm

• \( = \sum_{a=1}^{n} \sum_{b=a+1}^{n} E[ X_{a,b} ] \)

linearity of expectation

• \( = \sum_{a=1}^{n} \sum_{b=a+1}^{n} P( X_{a,b} = 1 ) \)

definition of expectation

• \( = \sum_{a=1}^{n} \sum_{b=a+1}^{n} \frac{2}{b - a + 1} \)

the reasoning we just did

• This is a big nasty sum, but we can do it.

• We get that this is less than \( 2n \ln(n) \).

Do this sum!

Ollie the over-achieving ostrich
Are we done?

• We saw that \( E[ \text{number of comparisons} ] = O(n \log(n)) \)
• Is that the same as \( E[ \text{running time} ] \)?

• In this case, **yes**.

• We need to argue that the running time is dominated by the time to do comparisons.

• (See CLRS for details).

• **QuickSort(A):**
  • If \( \text{len}(A) \leq 1 \):
    • return
  • Pick some \( x = A[i] \) at random. Call this the **pivot**.
  • **PARTITION** the rest of \( A \) into:
    • \( L \) (less than \( x \)) and
    • \( R \) (greater than \( x \))
  • Replace \( A \) with \( [L, x, R] \) (that is, rearrange \( A \) in this order)
  • **QuickSort(L)
  • **QuickSort(R)**
Worst-case running time

• Suppose that an adversary is choosing the “random” pivots for you.

• Then the running time might be $O(n^2)$ [on board]
  • In practice, this doesn’t usually happen.
A note on implementation

• This pseudocode is easy to understand and analyze, but is not a good way to implement this algorithm.

```
• QuickSort(A):
  • If len(A) <= 1:
    • return
  • Pick some x = A[i] at random. Call this the pivot.
  • PARTITION the rest of A into:
    • L (less than x) and
    • R (greater than x)
  • Replace A with [L, x, R] (that is, rearrange A in this order)
  • QuickSort(L)
  • QuickSort(R)
```

• Instead, implement it in-place (without separate L and R)
  • You may have seen this in 106b.
  • Here are some Hungarian Folk Dancers showing you how it’s done: https://www.youtube.com/watch?v=ywWBty6J5gz8
  • Check out IPython notebook for Lecture 5 for two different ways.
A better way to do Partition

Pivot

Initialize and Step forward.

When sees something smaller than the pivot, swap the things ahead of the bars and increment both.

Repeat till the end, then put the pivot in the right place.

See CLRS or Lecture 5 IPython notebook for pseudocode/real code.
QuickSort vs. smarter QuickSort vs. Mergesort?

- All seem pretty comparable...

See IPython notebook for Lecture 5

Hoare Partition is a different way of doing it, which you might have seen elsewhere. (You are not responsible for knowing it for this class).

The slicker in-place ones use less space, and also are a smidge faster on my system.
# QuickSort vs MergeSort

<table>
<thead>
<tr>
<th></th>
<th>QuickSort (random pivot)</th>
<th>MergeSort (deterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running time</strong></td>
<td>• Worst-case: $O(n^2)$</td>
<td>Worst-case: $O(n \log(n))$</td>
</tr>
<tr>
<td></td>
<td>• Expected: $O(n \log(n))$</td>
<td></td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>• Java for primitive types</td>
<td>• Java for objects</td>
</tr>
<tr>
<td></td>
<td>• C qsort</td>
<td>• Perl</td>
</tr>
<tr>
<td></td>
<td>• Unix</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• g++</td>
<td></td>
</tr>
<tr>
<td><strong>In-Place?</strong></td>
<td>Yes, pretty easily</td>
<td>Not easily* if you want to maintain both</td>
</tr>
<tr>
<td>(With $O(\log(n))$</td>
<td></td>
<td>stability and runtime.</td>
</tr>
<tr>
<td>extra memory)</td>
<td></td>
<td>(But pretty easily if you can sacrifice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>runtime).</td>
</tr>
<tr>
<td><strong>Stable?</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Other Pros</strong></td>
<td>Good cache locality if implemented for arrays</td>
<td>Merge step is really efficient with linked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lists.</td>
</tr>
</tbody>
</table>
Recap

• How do we measure the runtime of a randomized algorithm?
  • Expected runtime
  • Worst-case runtime

• QuickSort (with a random pivot) is a randomized sorting algorithm.
  • In many situations, QuickSort is nicer than MergeSort.
  • In many situations, MergeSort is nicer than QuickSort.

Code up QuickSort and MergeSort in a few different languages, with a few different implementations of lists A (array vs linked list, etc). What’s faster? (This is an exercise best done in C where you have a bit more control than in Python).
Next time

• Can we sort faster than $\Theta(n \log(n))$??

Before next time

• *Pre-lecture exercise* for Lecture 6.
  • Can we sort even faster than QuickSort/MergeSort?