Lecture 5
Randomized algorithms and QuickSort
Announcements

• HW3 is posted!
  • It’s a short one 😊

• Add/Drop deadline is Friday

• The alternate exam time has been determined
  • If you have not been told by us that your constraints can be accommodated (or have not been told by us that you will be told by us before the add/drop deadline) then you may not be able to take the final and you should drop this class.

• At the end of lecture, please wait until the actual end to start to pack up
  • The sound of 50 notebooks closing is actually pretty loud.
More announcements

• We will try to be more clear about expectations in HW.
• We will be providing latex templates starting with HW3.
  • There’s already a generic one in the “Resources” section of the website but we’ll make it tailored to each HW.
• We realize that Piazza answers could sometimes be more helpful.
  • We’re working on it!!
• We know lines are long at OH…
  • We are adding some Wednesday OH (check the calendar) and SCPD OH on Tuesday evenings.
  • Ways to help us out in OH:
    • Go to OH with a friend/group if you have similar questions: only one person from the group sign up for the queue!
    • Go to the homework party on Thursdays
    • Please use the map in queuestatus, and be descriptive about where you are. (Don’t write “Huang basement”)
    • Please respect the fact that CAs have to move on. They will be setting 15 minute timers.

• CAs in OH are not supposed to be “pre-grading.”
  • They can help you with the big idea, but please don’t ask them to pre-approve your solutions.
• Please do not call/text the CAs about CS161 stuff.
  • Talk to them in OH and on Piazza.
More announcements

• Some of your classmates are working on a project and ask that you fill out this poll (not associated with CS161).

• I’ll put a link up at the end of the lecture so you can fill it out then. (Or later on).
# Piazza Heroes

<table>
<thead>
<tr>
<th>Name, Email</th>
<th>number of answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Victor Li</td>
<td>14</td>
</tr>
<tr>
<td>Andrew Li</td>
<td>8</td>
</tr>
<tr>
<td>Jerry Me</td>
<td>7</td>
</tr>
<tr>
<td>Michaela M</td>
<td>6</td>
</tr>
<tr>
<td>Ilkyu Li</td>
<td>5</td>
</tr>
<tr>
<td>Noa B</td>
<td>5</td>
</tr>
<tr>
<td>Daniel Go</td>
<td>4</td>
</tr>
<tr>
<td>Dhruvik P</td>
<td>4</td>
</tr>
<tr>
<td>Evander D</td>
<td>4</td>
</tr>
<tr>
<td>Julie W</td>
<td>4</td>
</tr>
</tbody>
</table>
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...

[Chart showing time vs. size $n$ for different pivot selection methods, with annotations indicating 'We choose a pivot *cleverly*' and '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.

For today we will look at algorithms that always work and are probably fast.

e.g., **Select** with a random pivot is a randomized algorithm.
- Always works (aka, is correct).
- Probably fast.
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. You publish your algorithm.
2. Bad guy picks the input.
3. You run your randomized algorithm.

**Scenario 2**
1. You publish your algorithm.
2. Bad guy picks the input.
3. Bad guy chooses the randomness (fixes the dice) and runs your algorithm.

• 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...)
From your pre-lecture exercise:

**BogoSort**

- **BogoSort**(A)
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

Assume A has distinct entries

Suppose that you can draw a random integer in \(\{1, \ldots, n\}\) in time \(O(1)\). How would you randomly permute an array in-place in time \(O(n)\)?

Let \(X_i = \begin{cases} 1 & \text{if A is sorted after iteration } i \\ 0 & \text{otherwise} \end{cases}\)

- \(E[X_i] = \frac{1}{n!}\)
- \(E[\text{number of iterations until A is sorted}] = n!\)
1. Let $X$ be a random variable which is 1 with probability $1/100$ and 0 with probability $99/100$.
   
   a) $E[X] = 1/100$
   
   b) If $X_1, X_2, \ldots, X_n$ are iid copies of $X$, by linearity of expectation,
      $$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \frac{n}{100}$$
   
   c) Let $N$ be the index of the first 1. Then $E[N] = 100$.

To see part (c), either:

- You saw in CS109 that $N$ is a geometric random variable, and you know a formula for that.
- Suppose you do the first trial. If it comes up 1 (with probability $1/100$), then $N=1$. Otherwise, you start again except you’ve already used one trial. Thus:

   $$E[N] = \frac{1}{100} \cdot 1 + \left(1 - \frac{1}{100}\right) \cdot (1 + E[N]) = 1 + \left(1 - \frac{1}{100}\right)E[N]$$

   Solving for $E[N]$ we see $E[N] = 100$.

- (There are other derivations too).
Solutions to pre-lecture exercise 2

2. Let $X_i$ be 1 iff $A$ is sorted on iteration $i$.
   a) Okay. (There wasn’t actually a question for part (a)…)
   b) $E[X_i] = 1/n!$ since there are $n!$ possible orderings of $A$ and only one is sorted. (Suppose $A$ has distinct entries).
   c) Let $N$ be the index of the first 1. Then $E[N] = n!$.

Part (c) is similar to part (c) in exercise 1:
• You saw in CS109 that $N$ is a geometric random variable, and you know a formula for that. Or,
• Suppose you do the first trial. If it comes up 1 (with probability $1/n!$), then $N=1$. Otherwise, you start again except you’ve already used one trial. Thus:

$$E[N] = \frac{1}{n!} \cdot 1 + \left(1 - \frac{1}{n!}\right) \cdot (1 + E[N]) = 1 + \left(1 - \frac{1}{n!}\right) E[N]$$

Solving for $E[N]$ we see $E[N] = n!$
• (There are other derivations too).
From your pre-lecture exercise:

**BogoSort**

- **BogoSort**($A$)
  - **While** true:
    - Randomly permute $A$.
    - Check if $A$ is sorted.
    - **If** $A$ is sorted, **return** $A$.

- Let $X_i = \begin{cases} 1 & \text{if } A \text{ is sorted after iteration } i \\ 0 & \text{otherwise} \end{cases}$

- $E[X_i] = \frac{1}{n!}$

- $E[\text{number of iterations until } A \text{ is sorted}] = n!$
Expected Running time of BogoSort

\[ E[ \text{running time on a list of length } n ] \]
\[ = E[ (\text{number of iterations}) \times (\text{time per iteration}) ] \]
\[ = (\text{time per iteration}) \times E[\text{number of iterations}] \]
\[ = O(n \cdot n!) \]
\[ = \text{REALLY REALLY BIG.} \]
Worst-case running time of BogoSort?

Think-Pair-Share Terrapins!
1 minute: think
1 minute: pair and share

• **BogoSort**(A)
  • **While** true:
    • Randomly permute A.
    • Check if A is sorted.
    • **If** A is sorted, **return** A.
BogoSort

While true:
  Randomly permute A.
  Check if A is sorted.
  If A is sorted, return A.

Worst-case running time of BogoSort?

Infinite!

Think-Pair-Share Terrapins!

• BogoSort(A)
  • While true:
    • Randomly permute A.
    • Check if A is sorted.
    • If A is sorted, return A.
What have we learned?

• Expected running time:
  1. You publish your randomized algorithm.
  2. Bad guy picks an input.
  3. You get to roll the dice.

• Worst-case running time:
  1. You publish your randomized algorithm.
  2. Bad guy picks an input.
  3. Bad guy gets to “roll” the dice.

• Don’t use bogoSort.
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

• Expected runtime $O(n \log(n))$.

• Worst-case runtime $O(n^2)$.

• In practice works great!
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"

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

For the rest of the lecture, assume all elements of A are distinct.
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 elements of A are distinct. How would you change this if that’s not the case?
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:
  \[
  T(n) = O(n \log(n)).
  \]
The expected running time of QuickSort is \( O(n \log(n)) \).

**Proof:**

- \( E[|L|] = E[|R|] = \frac{n-1}{2} \).
  - The expected number of items on each side of the pivot is half of the things.
Aside

why is $E[|L|] = \frac{n-1}{2}$?

- $E[|L|] = E[|R|]$
  - by symmetry
- $E[|L| + |R|] = n - 1$
  - because $L$ and $R$ make up everything except the pivot.
- $E[|L|] + E[|R|] = n - 1$
  - By linearity of expectation
- $2E[|L|] = n - 1$
  - Plugging in the first bullet point.
- $E[|L|] = \frac{n-1}{2}$
  - Solving for $E[|L|]$. 

Remember, we are assuming all elements of $A$ are distinct.
The expected running time of QuickSort is $O(n \log(n))$.

**Proof:**

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

*Disclaimer: this proof is wrong.*
• Slow Sort(A):
  • If len(A) <= 1:
    • return
  • Pick the pivot x to be either max(A) or min(A), randomly
    • \ We can find the max and min in O(n) time
  • 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)
• Slow Sort(L)
• Slow Sort(R)

• Same recurrence relation:
  \[ T(n) = T(|L|) + T(|R|) + O(n) \]
• We still have \[ E[|L|] = E[|R|] = \frac{n-1}{2} \]
• But now, one of |L| or |R| is always n-1.
• You check: Running time is \( \Theta(n^2) \), with probability 1.

We can use the same argument to prove something false.
The expected running time of SlowSort is $O(n \log(n))$.

Proof:

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

*Disclaimer: this proof is wrong.*
What’s wrong?

• \( E[|L|] = E[|R|] = \frac{n-1}{2} \).
  • The expected number of items on each side of the pivot is half of the things.

• If that occurs, the running time is \( T(n) = O(n \log(n)) \).
  • Since the relevant recurrence relation is \( T(n) = 2T \left( \frac{n-1}{2} \right) + O(n) \)
  • Therefore, the expected running time is \( O(n \log(n)) \).

\textbf{That’s not how expectations work!}

• The running time in the “expected” situation is not the same as the expected running time.

• Sort of like how \( E[X^2] \) is not the same as \( (E[X])^2 \)
Instead

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

Next goal:

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

1. Pick 5 as a pivot
2. Partition on either side of 5
   - Recurse on [76] and pick 6 as a pivot.
   - Partition on either side of 6
     - Recurse on [7], it has size 1 so we’re done.
3. Recurse on [3142] and pick 3 as a pivot.
   - Partition around 3.
   - Recurse on [12] (done).
   - Recurse on [12] and pick 2 as a pivot.
     - Partition around 2.
     - Recurse on [1] (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?

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 the book)

- **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.
Counting comparisons

• The number of comparisons total during the algorithm is

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

• The expected number of comparisons is

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

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.

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[X_{a,b}]$$
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.

\[ \frac{2}{b - a + 1} \]

2 choices out of \( b-a+1 \)...
All together now...

Expected number of comparisons

- \( E\left[ \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} X_{a,b} \right] \)
- \( = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[X_{a,b}] \) linearity of expectation
- \( = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} P(X_{a,b} = 1) \) definition of expectation
- \( = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \frac{2}{b - a + 1} \) 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
Almost done

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

• In this case, yes.
• We need to argue that the running time is dominated by the time to do comparisons.
• See Lemma 5.2 in Algs. Illuminated.

• 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)
What have we learned?

• The expected running time of QuickSort is $O(n \log(n))$
Worst-case running time

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

• Then the running time might be $O(n^2)$
  • Eg, they’d choose to implement SlowSort
  • In practice, this doesn’t usually happen.
How should we implement this?

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

```plaintext
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=ywWBy6J5gz8
  • Check out IPython notebook for Lecture 5 for two different ways.
A better way to do Partition

Pivot

Choose it randomly, then swap it with the last one, so it’s at the end.

Initialize and Step forward.

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

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

See textbook 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 (c.f. CLRS Problem 7-1), which you might have seen elsewhere. You are not responsible for knowing it for this class.

In-place partition function uses less space, and also is a smidge faster on my system.
# QuickSort vs MergeSort

<table>
<thead>
<tr>
<th><strong>Running time</strong></th>
<th><strong>QuickSort (random pivot)</strong></th>
<th><strong>MergeSort (deterministic)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-case</td>
<td>$O(n^2)$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Expected</td>
<td>$O(n \log(n))$</td>
<td></td>
</tr>
</tbody>
</table>

- **Used by**
  - Java for primitive types
  - C `qsort`
  - Unix
  - g++
  - Java for objects
  - Perl

- **In-Place?**
  - Yes, pretty easily
  - Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice runtime.)

- **Stable?**
  - No
  - Yes

- **Other Pros**
  - Good cache locality if implemented for arrays
  - Merge step is really efficient with linked lists

---

*What if you want $O(n \log(n))$ worst-case runtime and stability? Check out “Block Sort” on Wikipedia!*

*These are just for fun. (Not on exam).*

*Understand this.*
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...)
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).

Ollie the over-achieving ostrich
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?
Some of your classmates are working on a project and ask that you fill out this poll (not associated with CS161): http://bit.ly/2TwXjcS