Lecture 5
Randomized algorithms and QuickSort
Announcements

• HW1 is graded!  Thanks TAs for super-fast turnaround!!
• HW2 is posted!  Due Friday.
• Please send any OAE letters to Jessica Su (stysu@stanford.edu) by Friday.
• Garrick attempts to make my cultural references more up-to-date:
Last time

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

We choose a pivot \textbf{randomly} and then a bad guy gets to decide what the array was.

We choose a pivot \textbf{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 \textbf{randomly}.
Randomized algorithms

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

e.g., Select with a random pivot is a randomized algorithm.

It was actually always correct
Looks like it’s probably fast but not always.
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)
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:

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 elements 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? (We’ll see later...)
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.
- If that occurs,
  - the running time is $T(n) = O(n \log(n))$.
- Therefore,
  - the expected running time is $O(n \log(n))$.

*Disclaimer: this proof is wrong.*
QuickSort(A):

• If len(A) <= 1:
  • return

• Pick the pivot x to be either max(A) or min(A), randomly
  • \[\text{We can find the max and min in O(n) time}\]

• 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)

• \textbf{Slow} Sort(L)
• \textbf{Slow} Sort(R)

• Same recurrence relation:
  \[T(n) = T(|L|) + T(|R|) + O(n)\]

• But now, one of |L| or |R| is n-1.
• Running time is O(n^2), with probability 1.
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 $T(n) = O(n \log(n))$.
- 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 $T(n) = O(n \log(n))$.
- Therefore, the expected running time is $O(n \log(n))$.

This argument says:

- $T(n) = \text{some function of } |L| \text{ and } |R|$ (✓)
- $\mathbb{E}[T(n)] = \mathbb{E}[\text{some function of } |L| \text{ and } |R|]$ (✓)
- $\mathbb{E}[T(n)] = \text{some function of } \mathbb{E}|L| \text{ and } \mathbb{E}|R|$ (✗)

That’s not how expectations work!
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

Pick 5 as a pivot

Partition on either side of 5

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

Partition on either side of 6

Recursively on [7], it has size 1, so we're 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 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 $P(X_{a,b} = 1) = \text{the probability that } a \text{ and } b \text{ are ever compared.}$

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.

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

2 choices out of \( b-a+1 \)...
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
Almost 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.

• 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)

• (See CLRS for details).
Conclusion

- 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.
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=ywWB4y6J5gZ8](https://www.youtube.com/watch?v=ywWB4y6J5gZ8)
  - Check out **IPython notebook** for Lecture 5 for two different ways.
A better way to do Partition

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

- All seem pretty comparable...

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

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.

See IPython notebook for Lecture 5
## QuickSort vs MergeSort

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

| Used by                   | • Java for primitive types                                   | • Java for objects                 |
|                           | • C qsort                                                    | • Perl                             |
|                           | • Unix                                                      |                                    |
|                           | • g++                                                       |                                    |

| In-Place? (With O(log(n)) extra memory) | 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 |

*In fact, I don’t know how to do this if you want O(n log(n)) worst-case runtime and stability.*
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).
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?
DEFINE HALFHEARTEDMERGESORT(list):
    IF LENGTH(list) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(list) / 2)  
    A = HALFHEARTEDMERGESORT(list[::PIVOT])
    B = HALFHEARTEDMERGESORT(list[PIVOT::])
    RETURN [A, B]  // HERE. SORRY.

DEFINE FASTBOGOSORT(list):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(list)):
        SHUFFLE(list)
        IF IISORTED(list):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

DEFINE JOBINTERVIEWQUICKSORT(list):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
        NO WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
        THE BIGGER ONES GO IN A NEW LIST
        THE EQUAL ONES GO INTO, UH
        THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
        THIS IS LIST A
        THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
        CALL IT LIST, OH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
        RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
        AM I ALLOWED TO USE THE STANDARD LIBRARIES?

DEFINE PANICSORT(list):
    IF IISORTED(list):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(list))
        LIST = LIST[PIVOT::] + LIST[:PIVOT]
    IF IISORTED(list):
        RETURN LIST
    IF IISORTED(list):
        RETURN LIST
    IF IISORTED(list):
        // THIS CAN'T BE HAPPENING
        RETURN LIST
    IF IISORTED(list):
        // COME ON COME ON
        RETURN LIST
        // OH JEEZ
        // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = []
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF /")
    SYSTEM("RM -RF ~/#")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") // PORTABILITY
    RETURN [1, 2, 3, 4, 5]