Lecture 5
Randomized algorithms and QuickSort
Announcements

• HW2 is posted! Due Friday.
• NO HOMEWORK assigned on Friday.
  • We only had one lecture this week...
• Updated induction slides from Lectures 3 and 4
  • Added bonus slides with different approaches
  • Clarified notation
• If you want examples of proofs by induction:
  • Handout about InsertionSort from Lecture 2
  • I just posted a handout for Lecture 4 with a formal proof that SELECT is correct.
  • CLRS
    • Recorded section from Week 1 went over induction
• Add/Drop Deadline Friday
  • We will try to get HW1 back before then.
Piazza Announcements

• Etiquette:
  • Please do not post public questions/comments which give away parts of answers, or your approach. Post a private question instead.

• Heroes!

<table>
<thead>
<tr>
<th>Name, Email</th>
<th>number of answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jabari Hastings</td>
<td>33</td>
</tr>
<tr>
<td>Ashish Paliwal</td>
<td>30</td>
</tr>
<tr>
<td>Trenton Chang</td>
<td>12</td>
</tr>
<tr>
<td>Brahm Capoor</td>
<td>10</td>
</tr>
<tr>
<td>Jiao Li</td>
<td>9</td>
</tr>
<tr>
<td>Magdy Saleh</td>
<td>8</td>
</tr>
<tr>
<td>Pranav Jain</td>
<td>8</td>
</tr>
<tr>
<td>Avery Wang</td>
<td>7</td>
</tr>
<tr>
<td>Julia Gong</td>
<td>7</td>
</tr>
<tr>
<td>Richard Lin</td>
<td>7</td>
</tr>
</tbody>
</table>
Please remind me

• Short break at 11:20.
  • If you have to leave at 11:20, please wait until then.

• Repeating student questions
  • If I don’t, holler “REPEAT THE QUESTION!”
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 \textit{cleverly}.

We choose a pivot \textit{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** $A$.

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

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

- $E[\text{number of iterations until } A \text{ is sorted}] = n!$
Solutions to pre-lecture exercise 1

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

\[ E[ \text{running time on a list of length } n ] \]
\[ = E[ (\text{number of iterations}) \times (\text{time per iteration}) ] \]
\[ = (\text{time per iteration}) \times E[\text{number of iterations}] \]
\[ = O(n \cdot n!) \]

= REALLY REALLY BIG.
Worst-case running time of BogoSort?

Think-Pair-Share Terrapins!

- **BogoSort**(A)
- **While** true:
  - Randomly permute A.
  - Check if A is sorted.
  - **If** A is sorted, **return** A.
Worst-case running time of BogoSort?

Think-Pair-Share Terrapins!

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

Infinite!
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: \textbf{QuickSort}

• Expected runtime $O(n\log(n))$.

• Worst-case runtime $O(n^2)$.

• In practice works great!
  • (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”

Arrange them like so:

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

Recurse on L and R:
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...)

IPython Lecture 5 notebook for actual code.
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.*
Red flag: 

We can use the same argument to prove something false.

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.
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 $O(n \log(n))$.

This argument says:

That’s not how expectations work!

$T(n) = \text{some function of } |L| \text{ and } |R| \checkmark$

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

$E[T(n)] = \text{some function of } E|L| \text{ and } E|R| \times$
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 [3142] and pick 3 as a pivot.

Partition around 3.

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

Partition around 2.

Recurse on [1] (done).

Recurse 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

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

- 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.
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 \( b-a+1 \)
Aside:
Why don’t we care about 1 and 7?

In a bit more detail:

• Let \( S = \{a,a+1,...,b\} \)

• \( P\{a, b \text{ are ever compared}\} \)
  
  \[ = \sum_{\text{stuff}} P\{a \text{ or } b \text{ picked first out of } S| \text{ stuff}\} \cdot P\{\text{stuff}\} \]
  
  where the sum is over all the stuff that does not involve \( S \).

• But since that stuff is independent of what happens with \( S \), this is equal to:

\[
= \sum_{\text{stuff}} P\{a \text{ or } b \text{ picked first out of } S\} \cdot P\{\text{stuff}\} \\
= P\{a \text{ or } b \text{ picked first out of } S\} \cdot \sum_{\text{stuff}} P\{\text{stuff}\} \\
= P\{a \text{ or } b \text{ picked first out of } S\} \\
= 2/|S| 
\]
Aside:

Why can we assume that the elements of the array are \( \{1,2,\ldots,n\} \)?

- More generally, say the elements of the array are \( a_1 < a_2 < \cdots < a_n \), so the array looks like:

\[
\begin{array}{ccccccc}
  a_7 & a_6 & a_3 & a_5 & a_1 & a_2 & a_4 \\
\end{array}
\]

- Then we’d do exactly the same thing, except we’d focus on the subscripts instead of the values. For example, the probability that \( a_2 \) and \( a_6 \) are ever compared is the probability that \( a_2 \) or \( a_6 \) are picked as a pivot before \( a_3, a_4, \) or \( a_5 \) are.
All together now...

**Expected number of comparisons**

- \( E\left[\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} X_{a,b}\right] \)
  
  This is the expected number of comparisons throughout the algorithm

- \( = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[X_{a,b}] \)
  
  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 \( 2n \ln(n) \).
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.

- (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)
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.
A note on implementation

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

• 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)

• 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](https://www.youtube.com/watch?v=ywWBy6J5gz8)
  • Check out IPython notebook for Lecture 5 for two different ways.
A better way to do Partition.

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 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.

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.

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? (With $O(\log(n))$ extra memory)</strong></td>
<td>Yes, pretty easily</td>
<td>Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice 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 lists</td>
</tr>
</tbody>
</table>
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 HALfHEArTED-MERGE-SOrT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALfHEArTED-MERGE-SOrT(LIST[:PIVOT])
    B = HALfHEArTED-MERGE-SOrT(LIST[PIVOT:])
    // UMMNNN
    RETURN [A, B] // HERE. SORRY.

DEFINe FAST-BOGO-SOrT(LIST):
    // AN OPTIMIZED BOGO-SORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF IS-SORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

DEFINe JOB-INTERVIEW-QUICK-SORT(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, UH, 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 PANIC-SOrT(LIST):
    IF IS-SORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[:PIVOT] + LIST[PIVOT:]
        IF IS-SORTED(LIST):
            RETURN LIST
        IF IS-SORTED(LIST):
            RETURN LIST:
            // THIS CAN'T BE HAPPENING
            RETURN LIST
        IF IS-SORTED(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]