Measuring and Predicting Running Time

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2024

Outline

Running times of different implementations

Running times of different implementations

- We have two implementations of PhoneDirectory: ArrayBasedPD and SortedPD.

Running times of different implementations

- We have two implementations of PhoneDirectory: ArrayBasedPD and SortedPD.
- Each has implementations of find, add, and remove.

Running times of different implementations

- We have two implementations of PhoneDirectory: ArrayBasedPD and SortedPD.
- Each has implementations of find, add, and remove.
- Can we compare their speeds?

lookUpEntry

lookUpEntry

- ArrayBasedPD.lookupEntry

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.lookupEntry

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.lookupEntry
  - Calls (SortedPD) find.

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.lookupEntry
  - Calls (SortedPD) find.
  - Ann, Bob, Eve, Ian, Jay, Zoe

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.lookupEntry
  - Calls (SortedPD) find.
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who does it compare Vic to?

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.lookupEntry
  - Calls (SortedPD) find.
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who does it compare Vic to?
  - Better but more helpful when n (size) is large.

lookUpEntry

- ArrayBasedPD.lookupEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Calls find.
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.lookupEntry
  - Calls (SortedPD) find.
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who does it compare Vic to?
  - Better but more helpful when n (size) is large.
  - Requires $\log_2 n$ comparisons

addOrChangeEntry

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.
  - find uses $\log_2 n$ comparisons

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.
  - Abe, Ann, Bob, Eve, Ian, Jay, Zoe

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.
  - Abe, Ann, Bob, Eve, Ian, Jay, Zoe
  - add uses n array accesses. Actually $n - 1$ reads and n writes, where n is 7. So $2n - 1$.

addOrChangeEntry

- ArrayBasedPD.addOrChangeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Has to call find and wait for find to finish (to make sure Vic isn't there already).
  - If you call addOrChangeEntry, you don't care how it does it, you just care how long it takes. No excuses, addOrChangeEntry!
  - find uses n comparisons
  - Then it calls add, which only takes 1 array access to add Vic to end of array.
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.
  - So n array access (actually $2n$) when array is full, but let's not worry about that now.
  - Total time is n comparisons to find plus 1 array access to add or change.
- SortedPD.addOrChangeEntry
  - Also has to call find and wait for find to finish.
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.
  - Abe, Ann, Bob, Eve, Ian, Jay, Zoe
  - add uses n array accesses. Actually $n-1$ reads and n writes, where n is 7. So $2n-1$.
  - Total time is $\log_2 n$ comparisons (find) plus $2n-1$ array accesses (add).

removeEntry

removeEntry

- ArrayBasedPD.removeEntry

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to "remove" Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who is the worst to remove?

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to "remove" Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who is the worst to remove?
  - Did you figure out it was Ann?

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who is the worst to remove?
  - Did you figure out it was Ann?
  - find takes $\log_2 n$ comparisons to locate Ann.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to "remove" Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who is the worst to remove?
  - Did you figure out it was Ann?
  - find takes $\log_2 n$ comparisons to locate Ann.
  - add takes n array reads and writes to move everyone else back.

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who is the worst to remove?
  - Did you figure out it was Ann?
  - find takes $\log_2 n$ comparisons to locate Ann.
  - add takes n array reads and writes to move everyone else back.
  - Bob, Eve, Ian, Jay, Zoe

removeEntry

- ArrayBasedPD.removeEntry
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Who takes longest to remove? Jay?
  - removeEntry calls find.
  - find takes 1 comparison to find Jay.
  - removeEntry calls remove.
  - Eve, Bob, Zoe, Ian, Ann
  - remove takes 2 array accesses to remove Jay.
  - Total time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - add still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons (find) and 2 array accesses (remove).
- SortedPD.removeEntry
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Who is the worst to remove?
  - Did you figure out it was Ann?
  - find takes $\log_2 n$ comparisons to locate Ann.
  - add takes n array reads and writes to move everyone else back.
  - Bob, Eve, Ian, Jay, Zoe
  - Total is $\log_2 n$ comparisons (find) and $2n$ array accesses (remove).

Summary

Summary

- ArrayBasedPD

Summary

- ArrayBasedPD
  - find: n comparisons

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)

Summary

- ArrayBasedPD
  - find: *n* comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons

Summary

- ArrayBasedPD
  - find: *n* comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: *n* comparisons
  - addOrChangeEntry: *n* comparisons plus 1 array access (usually)

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons
  - add: $2n$ array accesses

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons
  - add: $2n$ array accesses
  - remove: $2n$ array accesses

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons
  - add: $2n$ array accesses
  - remove: $2n$ array accesses
  - lookupEntry: $\log_2 n$ comparisons

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons
  - add: $2n$ array accesses
  - remove: $2n$ array accesses
  - lookupEntry: $\log_2 n$ comparisons
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses.

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 1 array access (usually)
  - remove: 2 array accesses
  - lookupEntry: n comparisons
  - addOrChangeEntry: n comparisons plus 1 array access (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons
  - add: $2n$ array accesses
  - remove: $2n$ array accesses
  - lookupEntry: $\log_2 n$ comparisons
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses.
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses.

Order Arithmetic

Order Arithmetic

- $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$

Order Arithmetic

- $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$
- Constants don't matter.

Order Arithmetic

- $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$
- Constants don't matter.
- $\log_2 n = 3.3219 \log_{10} n$, so we just say $\mathrm{O}(\log n)$

Order Arithmetic

- $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$
- Constants don't matter.
- $\log_2 n = 3.3219 \log_{10} n$, so we just say $\mathrm{O}(\log n)$
- Only the dominant term matters.

Order Arithmetic

- $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$
- Constants don't matter.
- $\log_2 n = 3.3219 \log_{10} n$, so we just say $\mathrm{O}(\log n)$
- Only the dominant term matters.
- Accurate, up to a constant factor, for large n.

Summary

Summary

- ArrayBasedPD

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $\mathrm{O}(n)$
  - add: 1 array access (usually) – $\mathrm{O}(1)$
  - remove: 2 array accesses – $\mathrm{O}(1)$
  - lookupEntry: n comparisons – $\mathrm{O}(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $\mathrm{O}(n) + \mathrm{O}(1) = \mathrm{O}(n)$
  - removeEntry: n comparisons plus 2 array accesses – $\mathrm{O}(n) + \mathrm{O}(1) = \mathrm{O}(n)$
- SortedPD

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $\mathrm{O}(n)$
  - add: 1 array access (usually) – $\mathrm{O}(1)$
  - remove: 2 array accesses – $\mathrm{O}(1)$
  - lookupEntry: n comparisons – $\mathrm{O}(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $\mathrm{O}(n) + \mathrm{O}(1) = \mathrm{O}(n)$
  - removeEntry: n comparisons plus 2 array accesses – $\mathrm{O}(n) + \mathrm{O}(1) = \mathrm{O}(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $\mathrm{O}(\log n)$
  - add: $2n$ array accesses – $\mathrm{O}(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.
  - Sorted add is (much) slower.

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.
  - Sorted add is (much) slower.
  - Sorted remove is (much) slower.

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.
  - Sorted add is (much) slower.
  - Sorted remove is (much) slower.
  - Sorted lookupEntry is (much) faster.

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.
  - Sorted add is (much) slower.
  - Sorted remove is (much) slower.
  - Sorted lookupEntry is (much) faster. Which is good, because that's probably what you do most.

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.
  - Sorted add is (much) slower.
  - Sorted remove is (much) slower.
  - Sorted lookupEntry is (much) faster. Which is good, because that's probably what you do most.
  - SortedPD addOrChangeEntry is the same.

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 1 array access (usually) – $O(1)$
  - remove: 2 array accesses – $O(1)$
  - lookupEntry: n comparisons – $O(n)$
  - addOrChangeEntry: n comparisons plus 1 array access (usually) – $O(n) + O(1) = O(n)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n) + O(1) = O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $2n$ array accesses – $O(n)$
  - remove: $2n$ array accesses – $O(n)$
  - lookupEntry: $\log_2 n$ comparisons – $O(\log n)$
  - addOrChangeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(\log n) + O(n) = O(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.
  - Sorted add is (much) slower.
  - Sorted remove is (much) slower.
  - Sorted lookupEntry is (much) faster. Which is good, because that's probably what you do most.
  - SortedPD addOrChangeEntry is the same.
  - SortedPD removeEntry is the same.

How to predict running time.

How to predict running time.

- So what use is $O(1)$ or $O(\log n)$, $O(n)$, or $O(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?

How to predict running time.

- So what use is $O(1)$ or $O(\log n)$, $O(n)$, or $O(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.

How to predict running time.

- So what use is $O(1)$ or $O(\log n)$, $O(n)$, or $O(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.

How to predict running time.

- So what use is $O(1)$ or $O(\log n)$, $O(n)$, or $O(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $O(n)$, we have

How to predict running time.

- So what use is $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, or $\mathrm{O}(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $\mathrm{O}(n)$, we have
  - $t = c \cdot n$

How to predict running time.

- So what use is $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, or $\mathrm{O}(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $\mathrm{O}(n)$, we have
  - $t = c \cdot n$
  - $10 = c \cdot 100$

How to predict running time.

- So what use is $O(1)$ or $O(\log n)$, $O(n)$, or $O(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $O(n)$, we have
  - $t = c \cdot n$
  - $10 = c \cdot 100$
  - $c = 1/10$

How to predict running time.

- So what use is $O(1)$ or $O(\log n)$, $O(n)$, or $O(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $O(n)$, we have
  - $t = c \cdot n$
  - $10 = c \cdot 100$
  - $c = 1/10$
- How long will it take for n=1000?

How to predict running time.

- So what use is $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, or $\mathrm{O}(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $\mathrm{O}(n)$, we have
  - $t = c \cdot n$
  - $10 = c \cdot 100$
  - $c = 1/10$
- How long will it take for n=1000?
  - $t = c \cdot n$

How to predict running time.

- So what use is $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, or $\mathrm{O}(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $\mathrm{O}(n)$, we have
  - $t = c \cdot n$
  - $10 = c \cdot 100$
  - $c = 1/10$
- How long will it take for n=1000?
  - $t = c \cdot n$
  - $t = 1/10 \cdot 1000$

How to predict running time.

- So what use is $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, or $\mathrm{O}(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $\mathrm{O}(n)$, we have
  - $t = c \cdot n$
  - $10 = c \cdot 100$
  - $c = 1/10$
- How long will it take for n=1000?
  - $t = c \cdot n$
  - $t = 1/10 \cdot 1000$
  - $t = 100$

How to predict running time.

- So what use is $\mathrm{O}(1)$ or $\mathrm{O}(\log n)$, $\mathrm{O}(n)$, or $\mathrm{O}(n \log n)$ if we don't know that constant, especially if it is a different constant in each case?
- We can measure the running time for one value of n and use that to extrapolate the running time for another value of n. Here is how to do it.
- Suppose ArrayBasedPD.find takes 10 microseconds for $n = 100$.
- Since the running time t is in $\mathrm{O}(n)$, we have
    - $t = c \cdot n$
    - $10 = c \cdot 100$
    - $c = 1/10$
- How long will it take for n=1000?
    - $t = c \cdot n$
    - $t = 1/10 \cdot 1000$
    - $t = 100$
- So the answer is 100 microseconds.

Another extrapolation

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $O(\log n)$, we have

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,
  - $t = c \cdot \log_{10} n$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,
  - $t = c \cdot \log_{10} n$
  - $t = 25 \cdot \log_{10} 1000$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,
  - $t = c \cdot \log_{10} n$
  - $t = 25 \cdot \log_{10} 1000$
  - $t = 25 \cdot 3$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,
  - $t = c \cdot \log_{10} n$
  - $t = 25 \cdot \log_{10} 1000$
  - $t = 25 \cdot 3$
  - $t = 75$

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,
  - $t = c \cdot \log_{10} n$
  - $t = 25 \cdot \log_{10} 1000$
  - $t = 25 \cdot 3$
  - $t = 75$
- So 75 microseconds.

Another extrapolation

- Now suppose SortedPD.find takes 50 microseconds for $n = 100$.
- More complicated methods often take longer for small n.
- This is the same reason that you don't drive your car to go next door.
- By the time you have opened the garage door, got in, started it up, etc., you will spend more time than just walking there.
- Since the running time is $\mathrm{O}(\log n)$, we have
  - $t = c \cdot \log_{10} n$
  - $50 = c \cdot \log_{10} 100$
  - $50 = c \cdot 2$
  - $c = 25$
- Even though the original analysis of binary search was for $\log_2 n$, I can use any base I want to because all logs differ by a constant factor.
- But you must use the *same* base for *every* log in the calculation.
- For $n = 1000$,
  - $t = c \cdot \log_{10} n$
  - $t = 25 \cdot \log_{10} 1000$
  - $t = 25 \cdot 3$
  - $t = 75$
- So 75 microseconds.
- Notice that I used the same log base 10. You can't switch log bases in the middle, or you will get a different (and wrong) answer.

- Here is the log base e version.

- Here is the log base e version.
- Calculate c from first n and t:

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:

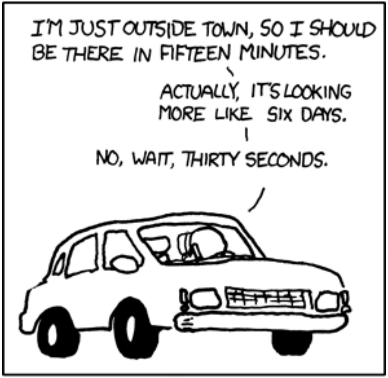
- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$
  - $t = 10.857 \cdot \ln 1000$

- Here is the log base *e* version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$
  - $t = 10.857 \cdot \ln 1000$
  - $t = 10.857 \cdot 6.9077$

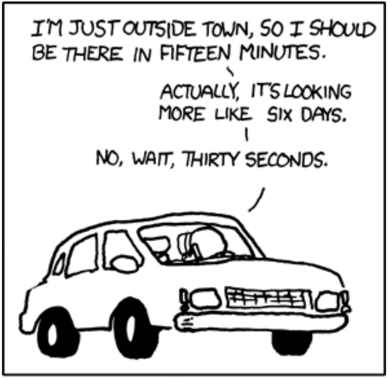
- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$
  - $t = 10.857 \cdot \ln 1000$
  - $t = 10.857 \cdot 6.9077$
  - $t = 74.997$

- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$
  - $t = 10.857 \cdot \ln 1000$
  - $t = 10.857 \cdot 6.9077$
  - $t = 74.997$
- Different log. Same answer!



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

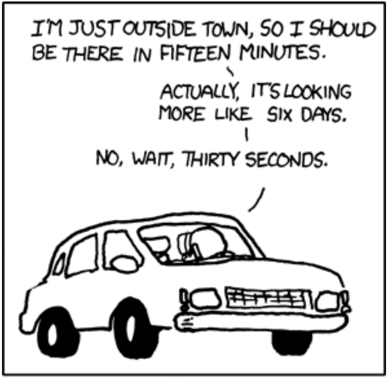
The author of the Windows file copy dialog responds!



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

The author of the Windows file copy dialog responds!

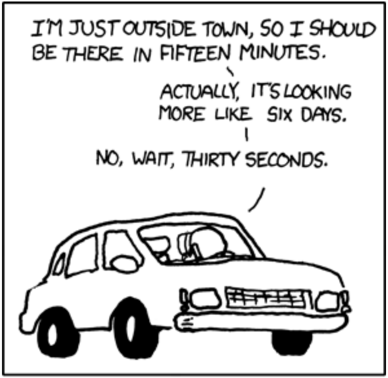
- Let's discuss this joke. In general the estimate is terrible and then gets better and better. Why?



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

The author of the Windows file copy dialog responds!

- Let's discuss this joke. In general the estimate is terrible and then gets better and better. Why?
- Let's say you do an experiment and it generates a number. It could be a time or a mass or anything. Just something you can measure. Unfortunately, the result is not very accurate. How can we increase the accuracy?



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

The author of the Windows file copy dialog responds!

- Let's discuss this joke. In general the estimate is terrible and then gets better and better. Why?
- Let's say you do an experiment and it generates a number. It could be a time or a mass or anything. Just something you can measure. Unfortunately, the result is not very accurate. How can we increase the accuracy?
- Answer: repeat the experiment many times and take the average.

How many times?

How many times?

- Suppose I am willing to spend one second timing my program.

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?
- Did you figure out 20,000 times?

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?
- Did you figure out 20,000 times?
- What is the general formula?

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?
- Did you figure out 20,000 times?
- What is the general formula?
- Run it for that many times and take the average.

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?
- Did you figure out 20,000 times?
- What is the general formula?
- Run it for that many times and take the average.
- Let's say it takes 1,010,203 microseconds seconds to run it 20,000 times.

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?
- Did you figure out 20,000 times?
- What is the general formula?
- Run it for that many times and take the average.
- Let's say it takes 1,010,203 microseconds seconds to run it 20,000 times.
- The average time 1010203 / 20000 = 50.51015 microseconds.

How many times?

- Suppose I am willing to spend one second timing my program.
- If the time for one run is 50 microseconds, how many times can I run it in one second?
- Did you figure out 20,000 times?
- What is the general formula?
- Run it for that many times and take the average.
- Let's say it takes 1,010,203 microseconds seconds to run it 20,000 times.
- The average time 1010203 / 20000 = 50.51015 microseconds.
- Much more accurate. We can trust 5 digits (maybe).

Summary

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- $\mathrm{O}()$ (order) notation simplifies all of these to $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$.

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- $\mathrm{O}()$ (order) notation simplifies all of these to $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$.
- The $\mathrm{O}()$ running time of a method on one input can be used to predict its running time on another input.

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- $\mathrm{O}()$ (order) notation simplifies all of these to $\mathrm{O}(1)$, $\mathrm{O}(\log n)$, or $\mathrm{O}(n)$.
- The $\mathrm{O}()$ running time of a method on one input can be used to predict its running time on another input.
- Accurate predictions can make or break a business and save millions of dollars.

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- $O()$ (order) notation simplifies all of these to $O(1)$, $O(\log n)$, or $O(n)$.
- The $O()$ running time of a method on one input can be used to predict its running time on another input.
- Accurate predictions can make or break a business and save millions of dollars.
- To improve the accuracy of a measurement, repeat it many times and take an average.

Summary

- ArrayBasedPD and SortedPD lookupEntry, addOrChangeEntry, and removeEntry take different amounts of time,
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- $O()$ (order) notation simplifies all of these to $O(1)$, $O(\log n)$, or $O(n)$.
- The $O()$ running time of a method on one input can be used to predict its running time on another input.
- Accurate predictions can make or break a business and save millions of dollars.
- To improve the accuracy of a measurement, repeat it many times and take an average.
- For example, run it once to get an approximate time. Figure out how many times you can run it in one second. Run it that many times and take the average running time.