Measuring and Predicting Running Time

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2016

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 removeEntry.

Running times of different implementations

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

find

find

- ArrayBasedPD.find

find

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

find

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

find

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

find

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

find

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

find

- ArrayBasedPD.find
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Look for Vic?
  - Have to compare Vic with n entries, where $n =$ size, which is 6.
- SortedPD.find
  - Only really helpful when n (size) is large.
  - Requires $\log_2 n$ comparisons

add

add

- ArrayBasedPD.add

add

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

add

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

add

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

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - Unless array is full, and then we need to allocate a bigger one, and copy everything over first.

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!
  - find uses $\log_2 n$ comparisons

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.
  - Abe, Ann, Bob, Eve, Ian, Jay, Zoe

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.
  - Abe, Ann, Bob, Eve, Ian, Jay, Zoe
  - n array accesses. Actually $n - 1$ reads and n writes, where n is 7. So $2n - 1$.

add

- ArrayBasedPD.add
  - Jay, Bob, Zoe, Ian, Ann, Eve
  - Add Vic?
  - Jay, Bob, Zoe, Ian, Ann, Eve, Vic
  - Only requires 1 array access to add Vic to end of array.
  - 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.
- SortedPD.add
  - Has to call find and wait for find to finish.
  - If you call add, you don't care how it does it, you just care how long it takes. No excuses, add!
  - find uses $\log_2 n$ comparisons
  - Ann, Bob, Eve, Ian, Jay, Zoe
  - Let's add Abe.
  - Abe, Ann, Bob, Eve, Ian, Jay, Zoe
  - n array accesses. Actually $n - 1$ reads and n writes, where n is 7. So $2n - 1$.
  - Total time is $\log_2 n$ comparisons plus $2n - 1$ array accesses.

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.
  - 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.
  - Eve, Bob, Zoe, Ian, Ann
  - 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.
  - Eve, Bob, Zoe, Ian, Ann
  - 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.
  - Eve, Bob, Zoe, Ian, Ann
  - 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to "remove" Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to "remove" Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Bob, Eve, Ian, Jay, Zoe
  - 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.
  - Eve, Bob, Zoe, Ian, Ann
  - Time for 1 comparison and 2 array accesses.
  - What about Eve? (Last entry)
  - Call to find takes n comparisons.
  - The program still uses 2 array accesses to “remove” Eve (but it could be smarter).
  - So Eve is worst case, requiring time for n comparisons and 2 array accesses.
- 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.
  - Bob, Eve, Ian, Jay, Zoe
  - n array reads and writes to move everyone else back.
  - Total is $\log_2 n$ comparisons and $2n$ array accesses. Actually the first n should be $n - 1$.

Summary

Summary

- ArrayBasedPD

Summary

- ArrayBasedPD
  - find: n comparisons

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 2 array accesses (usually)

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 2 array accesses (usually)
  - removeEntry: n comparisons plus 2 array accesses

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 2 array accesses (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 2 array accesses (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons

Summary

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

Summary

- ArrayBasedPD
  - find: n comparisons
  - add: 2 array accesses (usually)
  - removeEntry: n comparisons plus 2 array accesses
- SortedPD
  - find: $\log_2 n$ comparisons
  - add: $\log_2 n$ 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: 2 array accesses (usually) – $O(1)$

Summary

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

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 2 array accesses (usually) – $O(1)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n)$
- SortedPD

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 2 array accesses (usually) – $O(1)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 2 array accesses (usually) – $O(1)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $\log_2 n$ comparisons plus $2n$ array accesses – $O(n)$

Summary

- ArrayBasedPD
  - find: n comparisons – $O(n)$
  - add: 2 array accesses (usually) – $O(1)$
  - removeEntry: n comparisons plus 2 array accesses – $O(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $O(\log n)$
  - add: $\log_2 n$ comparisons plus $2n$ array accesses – $O(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $O(n)$

Summary

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

Summary

- ArrayBasedPD
  - find: n comparisons – $\mathrm{O}(n)$
  - add: 2 array accesses (usually) – $\mathrm{O}(1)$
  - removeEntry: n comparisons plus 2 array accesses – $\mathrm{O}(n)$
- SortedPD
  - find: $\log_2 n$ comparisons – $\mathrm{O}(\log n)$
  - add: $\log_2 n$ comparisons plus $2n$ array accesses – $\mathrm{O}(n)$
  - removeEntry: $\log_2 n$ comparisons plus $2n$ array accesses – $\mathrm{O}(n)$
- SortedPD compared to ArrayBasedPD
  - Sorted find is (much) faster.

Summary

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

Summary

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

Summary

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

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 $n = 100$.

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 $\mathrm{O}(1)$ 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 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?

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 $\mathrm{O}(\log n)$, 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:

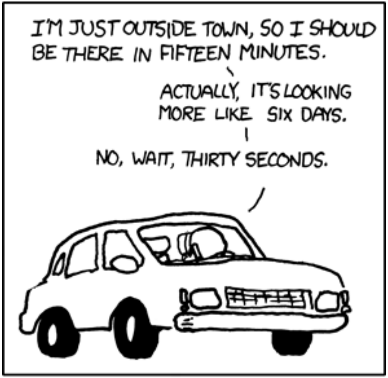
- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$

- 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:
  - $t = c \cdot \ln n$
  - $t = 10.857 \cdot \ln 1000$

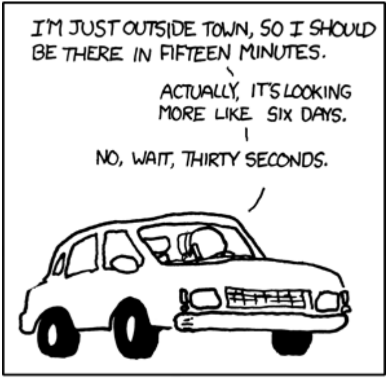
- Here is the log base e version.
- Calculate c from first n and t:
  - $t = c \cdot \ln n$
  - $50 = c \cdot \ln 100$
  - $50 = c \cdot 4.605$
  - $c = 10.857$
- Calculate t from second n:
  - $t = c \cdot \ln n$
  - $t = 10.857 \cdot \ln 1000$
  - $t = 10.857 \cdot 6.9077$

- 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!

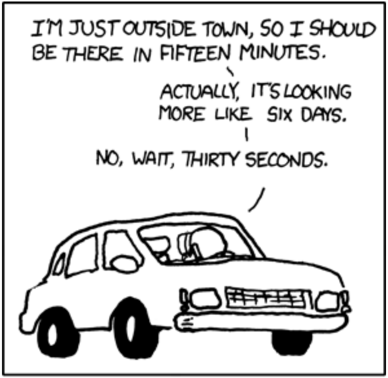


THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.



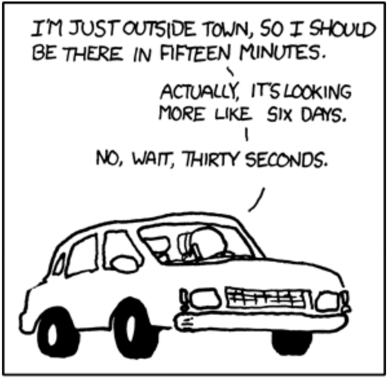
THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

- 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.

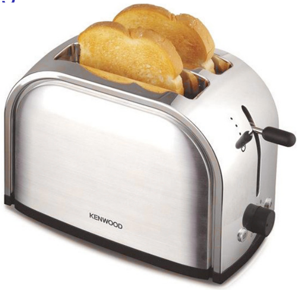
- Let's discuss this joke. In general the estimate is terrible and then gets better and better. Why?
- Let's say you do an experiment and it generates a number. It could be a time or a mass or anything. Just something you can measure. Unfortunately, the result is not very accurate. How can we increase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

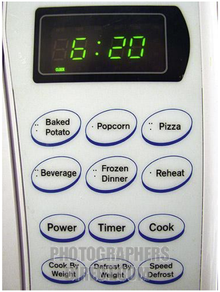


THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

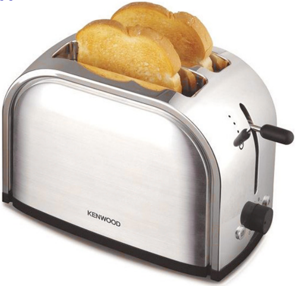
- Let's discuss this joke. In general the estimate is terrible and then gets better and better. Why?
- Let's say you do an experiment and it generates a number. It could be a time or a mass or anything. Just something you can measure. Unfortunately, the result is not very accurate. How can we increase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

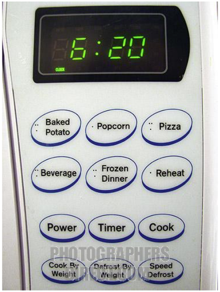
Timing the Toaster





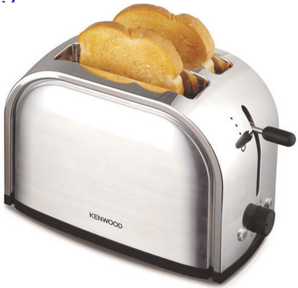
Timing the Toaster

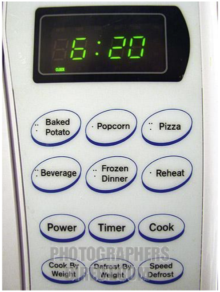




- I want to know how long my toaster takes (on setting 4–lightly browned),

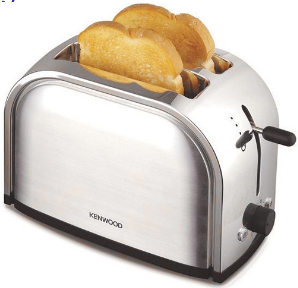
Timing the Toaster

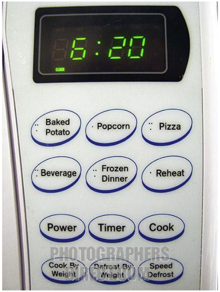




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.

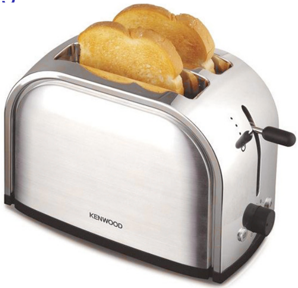
Timing the Toaster

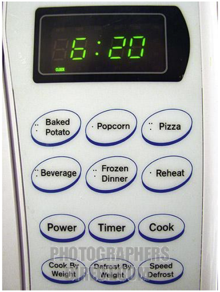




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.

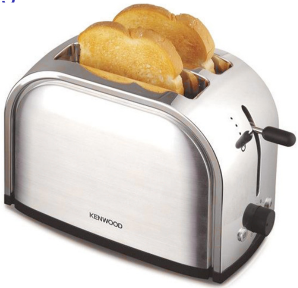
Timing the Toaster

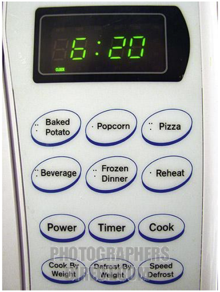




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?

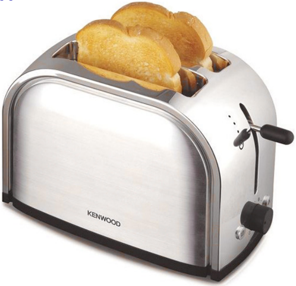
Timing the Toaster

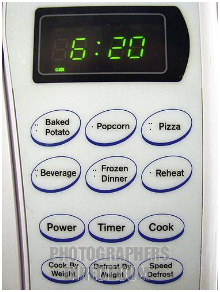




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?
  a. 6:22

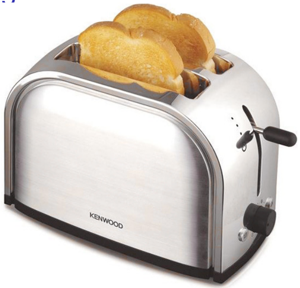
Timing the Toaster

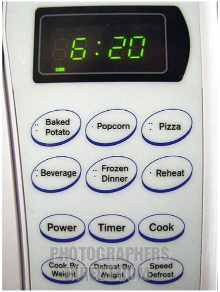




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?
  a. 6:22
  b. 6:23

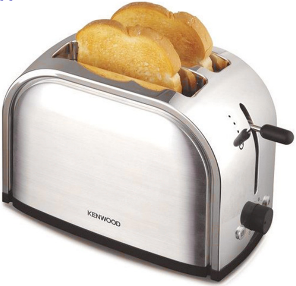
Timing the Toaster

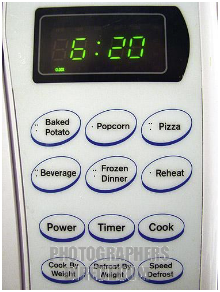




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?
  a. 6:22
  b. 6:23
  c. 6:22 and 6:23 are both possible

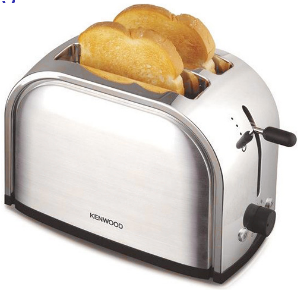
Timing the Toaster

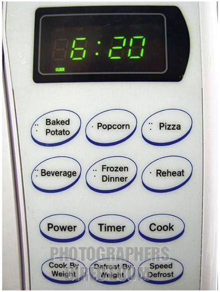




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?
  a. 6:22
  b. 6:23
  c. 6:22 and 6:23 are both possible
  d. 2:20

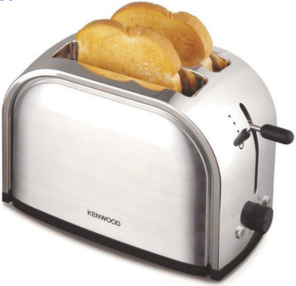
Timing the Toaster

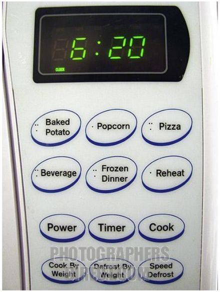




- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?
  a. 6:22
  b. 6:23
  c. 6:22 and 6:23 are both possible
  d. 2:20
- Answer: c.

Timing the Toaster





- I want to know how long my toaster takes (on setting 4–lightly browned),
- but I am in my pajamas, so the only clock I have is the digital clock on the microwave.
- The toaster in fact takes 2 minutes 20 seconds.
- I start the toaster when the clock says 6:20 (I could not find an image with 5:30 on it). What time will it say when the toaster pops?
  a. 6:22
  b. 6:23
  c. 6:22 and 6:23 are both possible
  d. 2:20
- Answer: c.
- It will say 6:22 with probability $2/3$ and 6:23 with probability $1/3$.

Increasing Accuracy

Increasing Accuracy

- With one test, I get the answer 2 minutes or 3 minutes.

Increasing Accuracy

- With one test, I get the answer 2 minutes or 3 minutes.
- I can increase the accuracy by running the experiment multiple times and taking the average.

trials	average
1	2.0
10	2.1
100	2.31
1000	2.328
10000	2.3438
100000	2.3365
1000000	2.333781
10000000	2.3333638
100000000	2.3333422
1000000000	2.333321892

Increasing Accuracy

- With one test, I get the answer 2 minutes or 3 minutes.
- I can increase the accuracy by running the experiment multiple times and taking the average.

trials	average
1	2.0
10	2.1
100	2.31
1000	2.328
10000	2.3438
100000	2.3365
1000000	2.333781
10000000	2.3333638
100000000	2.3333422
1000000000	2.333321892

- The Central Limit Theorem implies that averaging over n trials increases the accuracy by a factor of $\sqrt{n}$.

How many times?

How many times?

- Suppose I want three significant figures of accuracy.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.
  - So we want $1/\sqrt{n} < t/1000$.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.
  - So we want $1/\sqrt{n} < t/1000$.
  - So $n > 10^6/t^2$.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.
  - So we want $1/\sqrt{n} < t/1000$.
  - So $n > 10^6/t^2$.
- Example: $t = 0.1$ microseconds.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.
  - So we want $1/\sqrt{n} < t/1000$.
  - So $n > 10^6/t^2$.
- Example: $t = 0.1$ microseconds.
  - Set $n = 10^6/t^2 = 10^8$.

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.
  - So we want $1/\sqrt{n} < t/1000$.
  - So $n > 10^6/t^2$.
- Example: $t = 0.1$ microseconds.
  - Set $n = 10^6/t^2 = 10^8$.
  - Uh oh! How long with that take?

How many times?

- Suppose I want three significant figures of accuracy.
  - That means if the time is t, I want an error less than $t/1000$.
  - The theorem says that the error is $1/\sqrt{n}$ where n is the number of trials.
  - So we want $1/\sqrt{n} < t/1000$.
  - So $n > 10^6/t^2$.
- Example: $t = 0.1$ microseconds.
  - Set $n = 10^6/t^2 = 10^8$.
  - Uh oh! How long with that take?
  - $n \cdot t = 10^8 \cdot 0.1 = 10^7$ microseconds, which is 10 seconds.

Summary

Summary

- ArrayBasedPD and SortedPD find, add, and removeEntry take different amounts of time

Summary

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

Summary

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

Summary

- ArrayBasedPD and SortedPD find, add, and removeEntry take different amounts of time
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- Order ($\mathrm{O}()$) 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 find, add, and removeEntry take different amounts of time
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- Order ($\mathrm{O}()$) 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 find, add, and removeEntry take different amounts of time
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- Order ($\mathrm{O}()$) 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.
- To improve the accuracy of a measurement, repeat it many times and take an average.

Summary

- ArrayBasedPD and SortedPD find, add, and removeEntry take different amounts of time
- such as $\log_2 n$ comparisons plus $2n$ array accesses for SortedPD.removeEntry.
- Order ($\mathrm{O}()$) 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.
- To improve the accuracy of a measurement, repeat it many times and take an average.
- According to the Central Limit Theorem, if the time is t, we need to repeat it $n = 10^6/t^2$ times to get three significant figures of accuracy.