CSE 332

JUNE 30TH – AMORTIZED ANALYSIS

ADMINISTRATIVE

- Tokens

ADMINISTRATIVE

- **Tokens**
  - 3 tokens

ADMINISTRATIVE

- **Tokens**
  - 3 tokens
    - Opportunity to redo an exercise

ADMINISTRATIVE

- **Tokens**
  - 3 tokens
    - Opportunity to redo an exercise
    - One late day on a project

ADMINISTRATIVE

- **Tokens**
  - 3 tokens
    - Opportunity to redo an exercise
    - One late day on a project
- **Website should be all accurate now**

ADMINISTRATIVE

- **Tokens**
  - 3 tokens
    - Opportunity to redo an exercise
    - One late day on a project
- **Website should be all accurate now**
- **P1 Due Wednesday now**

TODAY

- **Recurrences**

TODAY

- Recurrences
- Master Theorem

TODAY

- Recurrences
- Master Theorem
- Amortized Analysis

TODAY

- Recurrences
- Master Theorem
- Amortized Analysis
- Project checkpoint

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior
- **Loops and iterations**

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior
- **Loops and iterations**
- **Recursive functions**

REVIEW

- **Algorithm Analysis**
  - Asymptotic behavior
- **Loops and iterations**
- **Recursive functions**
  - Recurrence relations

ANALYSIS

- **On Wednesday, we showed the formal recurrence approach**

ANALYSIS

- **On Wednesday, we showed the formal recurrence approach**
    - Break into recursive, non-recursive

ANALYSIS

- **On Wednesday, we showed the formal recurrence approach**
  - Break into recursive, non-recursive
  - Compute non-recursive computation time

ANALYSIS

- **On Wednesday, we showed the formal recurrence approach**
  - Break into recursive, non-recursive
  - Compute non-recursive computation time
  - Produce the recurrence

ANALYSIS

- **On Wednesday, we showed the formal recurrence approach**
  - Break into recursive, non-recursive
  - Compute non-recursive computation time
  - Produce the recurrence
  - Roll out the recurrence and produce the closed form

ANALYSIS

- **On Wednesday, we showed the formal recurrence approach**
  - Break into recursive, non-recursive
  - Compute non-recursive computation time
  - Produce the recurrence
  - Roll out the recurrence and produce the closed form
  - Upper-bound the closed form with bigO notation

ANALYSIS

- While this process is important, we can save some steps if all we care about is the upper bound

ANALYSIS

- **While this process is important, we can save some steps if all we care about is the upper bound**
  - bigO notation eliminates the need for constants

ANALYSIS

- **While this process is important, we can save some steps if all we care about is the upper bound**
  - bigO notation eliminates the need for constants
  - Lots of our messing around with c_0 and c_1 doesn't come through to the solution

ANALYSIS

- **While this process is important, we can save some steps if all we care about is the upper bound**
  - bigO notation eliminates the need for constants
  - Lots of our messing around with c_0 and c_1 doesn't come through to the solution

ANALYSIS

- **Merge sort**

ANALYSIS

- **Merge sort**
  - Separate the data into individual pieces

ANALYSIS

- **Merge sort**
  - Separate the data into individual pieces
  - Merge the pieces into larger and larger ones until the data is sorted

ANALYSIS

- **Merge sort**
  - Separate the data into individual pieces
  - Merge the pieces into larger and larger ones until the data is sorted
  - What is the recurrence here?

ANALYSIS

- **Merge sort**
  - Separate the data into individual pieces
  - Merge the pieces into larger and larger ones until the data is sorted
  - What is the recurrence here?
    - $T(n) = O(n) + 2T(n/2)$ for $n>1$

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$
  - $T(n) = c_0 + c_1 * n + 2T(n/2)$

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$
  - $T(n) = c_0 + c_1*n + 2T(n/2)$
  - $T(n) = c_0 + c_1*n + 2*c_0 + 2*c_1*n/2 + 4T(n/4)$

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$
  - $T(n) = c_0 + c_1*n + 2T(n/2)$
  - $T(n) = c_0 + c_1*n + 2*c_0 + 2*c_1*n/2 + 4T(n/4)$
  - $T(n) = 3*c_0 + 2*n*c_1 + 4T(n/4)$

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$
  - $T(n) = c_0 + c_1{*}n + 2T(n/2)$
  - $T(n) = c_0 + c_1{*}n + 2{*}c_0 + 2{*}c_1{*}n/2 + 4T(n/4)$
  - $T(n) = 3{*}c_0 + 2{*}n{*}c_1 + 4T(n/4)$
  - $T(n) = 3{*}c_0 + 2{*}n{*}c_1 + 4{*}c0 + 4{*}c1{*}n/4 + 8T(n/8)$

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$
  - $T(n) = c_0 + c_1 * n + 2T(n/2)$
  - $T(n) = c_0 + c_1 * n + 2 * c_0 + 2 * c_1 * n/2 + 4T(n/4)$
  - $T(n) = 3 * c_0 + 2 * n * c_1 + 4T(n/4)$
  - $T(n) = 3 * c_0 + 2 * n * c_1 + 4 * c0 + 4 * c1 * n/4 + 8T(n/8)$
  - …

ANALYSIS

- **Merge sort**
  - $T(n) = O(1)$ for $n < 2$
  - $T(n) = O(n) + 2T(n/2)$ for $n>1$
  - $T(n) = c_0 + c_1{*}n + 2T(n/2)$
  - $T(n) = c_0 + c_1{*}n + 2{*}c_0 + 2{*}c_1{*}n/2 + 4T(n/4)$
  - $T(n) = 3{*}c_0 + 2{*}n{*}c_1 + 4T(n/4)$
  - $T(n) = 3{*}c_0 + 2{*}n{*}c_1 + 4{*}c0 + 4{*}c1{*}n/4 + 8T(n/8)$
  - …
  - We can derive the pattern this way, but it isn't necessarily intuitive

ANALYSIS

- **Merge sort**
  - Consider the problem graphically

ANALYSIS

- **Merge sort**
  - Consider the problem graphically
  - Each recursive call is a node in a tree

ANALYSIS

- **Merge sort**
  - Consider the problem graphically
  - Each recursive call is a node in a tree
  - Helpful for logarithmic patterns and when each run calls itself more than once

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions
  - If $T(n) = a*T(n/b)+n^c$ for $n > n_0$ and if the base case is a constant

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions
  - If $T(n) = a*T(n/b)+n^c$ for $n > n_0$ and if the base case is a constant
    - Case 1: $\log_b(a) < c$: $T(n) = O(n^c)$
    - Case 2: $\log_b(a) = c$: $T(n) = O(n^c \lg n)$
    - Case 3: $\log_b(a) > c$: $T(n) = O(n^{\log a})$

ANALYSIS

- **Master theorem**
  - These recurrences all follow a similar pattern
  - Therefore, if you can produce a recurrence, there is actually a procedural way to produce solutions
  - If $T(n) = a*T(n/b)+n^c$ for $n > n_0$ and if the base case is a constant
    - Case 1: $\log_b(a) < c$: $T(n) = O(n^c)$
    - Case 2: $\log_b(a) = c$: $T(n) = O(n^c \lg n)$
    - Case 3: $\log_b(a) > c$: $T(n) = O(n^{\log a})$
  - Verify with merge sort: $a = 2$, $b = 2$, $c = 1$

ANALYSIS

- Recurrences come up all the time

ANALYSIS

- **Recurrences come up all the time**
  - Analyze methods and iterative approaches through the normal methods

ANALYSIS

- **Recurrences come up all the time**
  - Analyze methods and iterative approaches through the normal methods
  - Recursive functions use a recurrence

ANALYSIS

- **Recurrences come up all the time**
  - Analyze methods and iterative approaches through the normal methods
  - Recursive functions use a recurrence
  - Possible to get to bigO solution quickly

ANALYSIS

- **Recurrences come up all the time**
  - Analyze methods and iterative approaches through the normal methods
  - Recursive functions use a recurrence
  - Possible to get to bigO solution quickly
  - Usually for worst-case analysis

ANALYSIS

- Final analysis type

ANALYSIS

- **Final analysis type**
  - Worst-case

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array
    - Resizing is the costly O(n) operation

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array
    - Resizing is the costly O(n) operation
    - This occurs in predictable ways

ANALYSIS

- **Final analysis type**
  - Worst-case
    - Consider adding to an unsorted array
    - Resizing is the costly O(n) operation
    - This occurs in predictable ways
    - Do these types of operations really slow down the function?

AMORTIZED ANALYSIS

- Adding to unsorted array

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add n elements into the array?

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add n elements into the array?
  - Let's say the array is full with n elements and we add n more

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)
  - Amortized over the whole set of operations, each one is only O(1) time

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)
  - Amortized over the whole set of operations, each one is only O(1) time
  - What does this depend on?

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - How long does it take to add *n* elements into the array?
  - Let's say the array is full with *n* elements and we add *n* more
  - It takes n-1*O(1) + 1*O(n) = O(n)
  - Amortized over the whole set of operations, each one is only O(1) time
  - What does this depend on?
    - **Doubling the array**

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - What if we only add some constant number to the array?
  - Let's resize and add 10,000 elements every time
  - How long does it take to add *n* elements?
  - n-n/10,000*O(1) + n/10,000*O(n)

AMORTIZED ANALYSIS

- **Adding to unsorted array**
  - What if we only add some constant number to the array?
  - Let's resize and add 10,000 elements every time
  - How long does it take to add *n* elements?
  - $n - n/10{,}000 * O(1) + n/10{,}000 * O(n) = O(n^2)$
  - This is for any constant, regardless of how large

P1 CHECKPOINT 1

- Get into your project groups

P1 CHECKPOINT 1

- Get into your project groups
- We’ll be coming around to meet with you for a few minutes

P1 CHECKPOINT 1

- **Get into your project groups**
- **We'll be coming around to meet with you for a few minutes**
  - Have you made it through part 1?
  - How has the team been working?
  - Have you started part 2?

P1 CHECKPOINT 1

- **Get into your project groups**
- **We'll be coming around to meet with you for a few minutes**
  - Have you made it through part 1?
  - How has the team been working?
  - Have you started part 2?
- **Good opportunity to make sure everything is on the right track**

P1 CHECKPOINT 1

- **Get into your project groups**
- **We'll be coming around to meet with you for a few minutes**
  - Have you made it through part 1?
  - How has the team been working?
  - Have you started part 2?
- **Good opportunity to make sure everything is on the right track**
- **Once one of us has talked with you, you're free to go**

NEXT WEEK

- Dictionaries

NEXT WEEK

- **Dictionaries**
  - Binary Search Trees
  - Balancing