University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 7: this Friday @ 11:59 pm
  - Lab 6: next Monday 10/31 @ 11:59 pm
  - **Assignment 2: Monday 11/7 @ 11:59 pm**
- Midterm Feedback session
  - next lecture
- Opportunity for correcting your Midterm answers
  - within a point budget
- Live Support Session for Assignment 2
  - This Friday 10/28 @ 5:30 pm
- Lab Solutions for Lab 1-Lab 5 available on the handouts repository
- Live QA Session on Piazza every Friday 4:30-5:30 pm

Previous Lecture …

- More recursion examples
  - Fibonacci numbers
  - linear search
- Recursion tree analysis
- Recursion may lead to poor solutions
- Average running time analysis

Today …

- Amortized running-time analysis
- More recursion examples
  - binary search
  - finding words in a grid of letters
- Recursion analysis
  - one more example

Amortized Analysis

Average over a sequence of operations

Amortized Analysis Example: add(T) of ArrayList

- Recall that this version of the method adds to the end of the list
- O(1) time: We can go directly to the last location and insert there
- Runtime for Resizable Array?
- Is it also O(1)?
- Some adds take significantly more time, why?
  - we have to first allocate a new array **and copy all of the data** into it
  - O(n) time
  - So, when resizing happens we have O(n) + O(1) → O(n) total time
- So, is add O(1) or O(n)?
- Amortized analysis can help here

Amortized Analysis Example: add(T) of ArrayList

- So, we have an operation that sometimes takes O(1) and sometimes takes O(n)
- How do we handle this issue?
- Amortized Time (see http://en.wikipedia.org/wiki/Amortized_analysis)
  - Average time required over a sequence of operations
  - Individual operations may vary in their run-time, but we can get a consistent time for the overall sequence
  - Let's stick with the add() method for resizable array list and consider 2 different options for resizing:
    1) Increase the array size by 1 each time we resize
    2) Double the array size each time we resize (which is the way the authors actually did it)

Amortized Analysis Example: add(T) of ArrayList

1) Increase the array size by 1 each time we resize

- Note that with this approach, once we resize we will have to do it with every add
- Thus, rather than O(1) our add() is now O(n) **all the time**
- Specifically, **assume the initial array size is 1**
  - On insert 1 we just add the item (1 assignment)
  - On insert 2 we allocate and assign 2 items
  - On insert 3 we allocate and assign 3 items
  - …
  - Overall, **for n add() ops,** the total number of assignments we have to make:

    $1 + 2 + 3 + \ldots + n =$

    $n(n+1)/2 \rightarrow O(n^2)$

- Since we did n add() operations overall, our amortized time is $O(n^2)/n = O(n)$

Amortized Analysis Example

2) Double the array size each time we resize (assume the initial array size is 1)

Insert #	# of assignments	End array size
1	1	1
2	$2 = 1 + 1 = 1 + 2^0$	2
3	$3 = 1 + 2^1$	4
4	1	4
5	$5 = 1 + 2^2$	8
…	1	8
9	$9 = 1 + 2^3$	16
…	1	16
17	$17 = 1 + 2^4$	32
…	1	32
32	1	32

Blue is cost of actually adding the item

Red is cost of copying data to a new array

Amortized Analysis Example

- Note that every row has 1 assignment (blue)
- Rows that are $2^k + 1$ for some k have an additional 2^k assignments (red) to copy data
- So, for n **add ops**, we have a total of
  - n assignments for the actual add
  - $2^0 + 2^1 + \ldots + 2^x$ for the copying
  - What is *x?*
  - *The largest value such that* $2^x + 1 <= n$
  - $x = \text{floor}(\lg_2(n-1))$
  - Total number of red assignments = $2^0 + 2^1 + \ldots + 2^x$

$$= \sum_{i=0}^{lg_2(n-1)} 2^i$$

$$= \frac{first\ term(1-r^n)}{1-r}$$

$$= \frac{1(1-2^{\lg n})}{1-2}$$

$$= \frac{1(1-n)}{-1} = n - 1$$

Insert #	# of assignments	End array size
1	1	1
2	2 = 1 + 1 = 1 + 2^0	2
3	3 = 1 + 2^1	4
4	1	4
5	5 = 1 + 2^2	8
…	1	8
9	9 = 1 + 2^3	16
…	1	16
17	17 = 1 + 2^4	32
…	1	32
32	1	32

Amortized Analysis

- Total number of assignments is n + (n-1) = 2n-1 → O(n)
- Since we did n add() operations overall, our amortized time is O(n)/n = O(1), a constant
- Recall that when increasing by 1 we had O(n) amortized time
  - Note how much better our performance is when we double the array size
- Ok, that one was a bit complicated
  - Had a good deal of math in it
  - But that is what algorithm analysis is all about
  - If you can do some math you can save yourself some programming!

Amortized vs. Average-case Analysis

- Amortized running-time is the average running-time over a **sequence of operations**
  - useful when the operations have varying running-time
  - e.g., add() is O(1) without array resizing and O(n) when resizing happens
- Average-case running-time is the average over a **probability distribution** of the input
  - Typically, a uniform distribution is assumed
  - e.g., assume all array positions have the same probability of holding the target item in search

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a,                    desiredItem)
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else // desiredItem > a[mid]
```

Binary Search of a Sorted Array

Recursive algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else // desiredItem > a[mid]
   return binarySearch(a, mid + 1, last, desiredItem)
```

Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that does not find its target

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$16 > 10$, so search the right half of the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that does not find its target

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$16 > 10$, so search the right half of the array.

Look at the middle entry, 18:

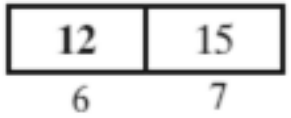
12	15	**18**	21	24	26
6	7	8	9	10	11

$16 < 18$, so search the left half of the array.

Binary Search of a Sorted Array

A recursive binary search of a sorted array that does not find its target

Look at the middle entry, 12:

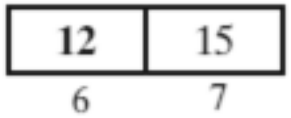


16 > 12, so search the right half of the array.

Binary Search of a Sorted Array

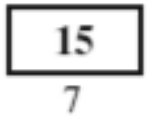
A recursive binary search of a sorted array that does not find its target

Look at the middle entry, 12:



16 > 12, so search the right half of the array.

Look at the middle entry, 15:

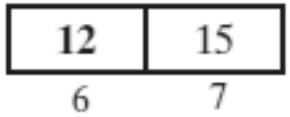


16 > 15, so search the right half of the array.

Binary Search of a Sorted Array

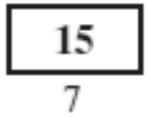
A recursive binary search of a sorted array that does not find its target

Look at the middle entry, 12:



16 > 12, so search the right half of the array.

Look at the middle entry, 15:



16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Time complexity of Binary Search

- To simplify calculations, we'll cheat a bit:
  1) Assume that the array is cut exactly in half with each iteration
     - In reality, it may vary by one element either way
  2) Assume that the initial size of the array, n, is an exact power of 2, or 2^k for some k
     - In reality, it can be any value
     - However, it will not affect our results
- Ok, so we have the following:
  - At call 0: $n_0 = 2^k$
  - At call 1, $n_1 = n_0/2 =$ (in terms of k?)
  - $= 2^{k-1}$
  - At call 2, $n_2 = n_1/2 =$ (in terms of k?)
  - $= 2^{k-2}$
  - At call i, $n_i = n_{i-1}/2 =$ (in terms of k?)
  - $= 2^{k-i}$
  - Last call is when n = 1 = (in terms of k?)
  - $= 2^0$

Time complexity of Binary Search

- How many calls are needed?
- In the last call, k-i = 0 → i = k
- We do one comparison (test) per call
- Thus, we have a total of k+1 comparisons maximum
  - But, $n = 2^k$
  - So k = ?
  - $k = \lg_2 n$
  - which makes k + 1 = ?
  - $= \lg_2 n + 1$
- This leads to our final answer of ?
- $O(\lg_2 n)$

Binary Search of a Sorted Chain

- To find the middle of the chain you must traverse the whole chain
- Then must traverse one of the halves to find the middle of that half
- Hard to implement
- Less efficient than sequential search!

Choosing between Sequential Search and Binary Search

The time efficiency of searching, expressed in Big Oh notation

	Best Case	Average Case	Worst Case
Sequential search (unsorted data)	O(1)	O(n)	O(n)
Sequential search (sorted data)	O(1)	O(n)	O(n)
Binary search (sorted array)	O(1)	O(log n)	O(log n)

Word Search Problem

- Find all words in a grid of letters
  - horizontally, vertically, diagonally

B	Y	S	Y	U	S	M	C	X	E	G	A	M	E	Y	O	Y	O	Y	Y
B	E	J	D	L	A	O	N	N	F	V	S	R	I	N	E	V	U	O	S
E	B	Y	L	R	I	B	I	S	E	R	U	T	A	I	N	I	M	H	A
A	Z	O	B	N	A	Z	L	I	C	N	E	P	R	E	K	C	I	T	S
D	D	L	S	T	A	C	D	B	R	L	A	J	G	I	H	R	F	R	C
S	E	B	Q	G	R	R	U	I	O	C	A	B	L	P	S	A	D	V	S
S	F	K	A	Y	O	I	B	G	E	O	A	D	A	T	C	E	R	S	E
E	P	M	E	C	M	B	H	L	N	S	K	R	E	O	L	B	O	E	A
V	R	I	K	Y	O	G	T	S	E	I	G	S	M	M	L	D	C	V	S
C	E	P	T	N	Y	T	R	B	S	O	Y	I	Z	O	O	Y	E	A	H
A	C	U	S	R	O	C	A	T	T	E	C	A	K	H	D	D	R	E	E
L	I	P	C	B	O	L	H	U	C	N	E	B	L	C	D	D	M	L	L
E	P	P	F	H	L	P	A	A	B	E	A	T	A	P	R	E	U	S	L
N	E	E	V	C	I	F	H	O	I	L	S	N	Z	U	E	T	G	P	V
D	Q	T	A	P	L	N	O	Y	L	N	S	N	N	N	P	L	S	M	K
A	U	R	W	A	M	K	A	S	S	Y	E	K	I	E	A	V	M	A	N
R	D	Z	G	A	Y	R	L	E	W	E	J	B	E	M	P	A	E	T	A
A	N	S	P	O	S	T	E	R	B	O	O	K	M	A	R	K	S	S	B

Backtracking Framework

```
void solve(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
          undo changes to partial solution
      }
   }
}
```

Backtracking Algorithm for Word Search

```
void solve(row and column of current cell, word string so far) {
   for each of the eight directions { //why 8?
      if neighbor down the direction is in the board {
         append letter to word string
         if word string is in the dictionary
            add word string to set of solutions
         if word string is a prefix
            solve(row and column of neighbor, word string)
            delete last letter of word string
      }
   }
}
```