

Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Today's Topics

Recursion Week continues!

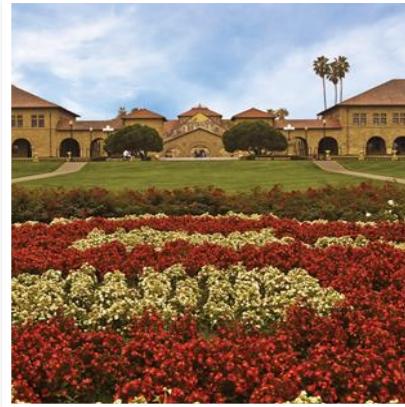
- Today, two applications of recursion:
 - › Binary Search (one of the fundamental algorithms of CS)
 - We saw the idea of this on Wed, but today we'll code it up
 - Callback to Big-O discussion
 - › Generating sequences
 - *cough* Assignment 3 *cough*

Next time:

- More recursion! It's Recursion Week!
- Like Shark Week, but more nerdy

Binary Search Refresher

(RECALL FROM WEDNESDAY'S
LECTURE)



Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- **Efficiency Hack: Jump to the middle of the Vector and look there to find:**

- › X (answer Yes)
- › A number greater than X (rule out entire second half of Vector)

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- › A number less than X (rule out entire first half of Vector)

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- Key observation: with **one** comparison, you ruled out **N/2** of the N cells in the Vector!

Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- **Extreme Efficiency Hack: Keep jumping to the middle!**

- › Let's say our first jump to the middle found a number less than X, so we ruled out the whole first half:

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- › Now jump to the middle of the remaining second half:

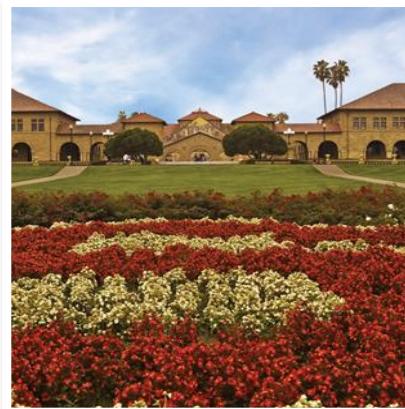
0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- Key observation: we do one piece of work, then delegate the rest. **Recursion!!**

Binary Search Implementation

NOW WE UNDERSTAND THE
APPROACH.

WHAT DOES THE CODE LOOK
LIKE?





From
previous
lecture

The recursive function pattern

Always two parts:

Base case:

- This problem is so tiny, it's hardly a problem anymore! Just give answer.

Recursive case:

- This problem is still a bit large, let's (1) bite off just one piece, and (2) delegate the remaining work to recursion.

Translated to code

```
int factorial(int n) {  
    if (n == 1) { // Easy! Return trivial answer  
        return 1;  
    } else { // Not easy enough to finish yet!  
        return n * factorial(n - 1);  
    }  
}
```

Binary Search pseudocode

- We'll write the real C++ code together on Friday, but here's the outline/pseudocode of how it works:

```
bool binarySearch(Vector<int>& data, int key)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key);
    } else {
        return binarySearch(data[second half only], key);
    }
}
```

Base case: we shrank the search problem so tiny it no longer exists!

Recursive case:

Do one piece of work (comparison)

Delegate the rest of the work

```
bool binarySearch(Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```

```
bool binarySearch(Vector<int>& data, int key, int start, int end) {
```

Recursive Function Design Tip: Wrapper function

- When we want to write a recursive function that needs more book-keeping data passed around than an outsider user would want to worry about, do this:
 1. Write the function as you need to for correctness, using any extra book-keeping parameters you like, in whatever way you like.
 2. Make a second function that the outside world sees, using only the minimum number of parameters, and have it do nothing but call the recursive one.
 - Called a “wrapper” function because it’s like pretty outer packaging.



```
bool binarySearch(Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```



```
bool binarySearch(Vector<int>& data, int key, int start, int end) {  
    if (start > end) {  
        return false;  
    }  
    int mid = (start + end) / 2;  
    if (key == data[mid]) {  
        return true;  
    } else if (key < data[mid]) {  
        return binarySearch(data, key, _____, _____);  
    } else {  
        return binarySearch(data, key, _____, _____);  
    }  
}
```

```
bool binarySearch(Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```



```
bool binarySearch(Vector<int>& data, int key, int start, int end) {  
    if (start > end) {  
        return false;  
    }  
    int mid = (start + end) / 2;  
    if (key == data[mid]) {  
        return true;  
    } else if (key < data[mid]) {  
        return binarySearch(data, key, _____, _____);  
    } else {  
        return binarySearch(data, key, _____, _____);  
    }  
}
```

Your Turn:

What goes on the blanks below, to divide the remaining searchable region of our vector in half?

```
bool binarySearch(const Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```



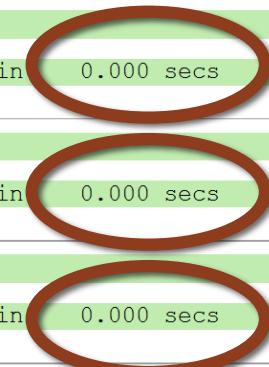
```
bool binarySearch(const Vector<int>& data, int key, int start, int end) {  
    if (start > end) {  
        return false;  
    }  
    int mid = (start + end) / 2;  
    if (key == data[mid]) {  
        return true;  
    } else if (key < data[mid]) {  
        return binarySearch(data, key, start, mid - 1);  
    } else {  
        return binarySearch(data, key, mid + 1, end);  
    }  
}
```

Binary Search performance

```
SimpleTest BinarySearch
Tests from PROVIDED_TEST

Correct (PROVIDED_TEST, binsearch.cpp:88) Basic correctness: found value
Correct (PROVIDED_TEST, binsearch.cpp:93) Basic correctness: missing value
Correct (PROVIDED_TEST, binsearch.cpp:98) Edge case: found first value
Correct (PROVIDED_TEST, binsearch.cpp:103) Edge case: found last value
Correct (PROVIDED_TEST, binsearch.cpp:108) Timing on 10K elements
    Line 112 TIME_OPERATION binarySearch(data, 5) (size = 10000) completed in 0.000 secs
Correct (PROVIDED_TEST, binsearch.cpp:115) Timing on 100K elements
    Line 119 TIME_OPERATION binarySearch(data, 5) (size = 100000) completed in 0.000 secs
Correct (PROVIDED_TEST, binsearch.cpp:122) Timing on 1M elements
    Line 126 TIME_OPERATION binarySearch(data, 5) (size = 1000000) completed in 0.000 secs
Passed 7 of 7 tests. Great!
```

Q. We saw the test take a long time to run for 1M, but it reports 0.000 secs. What's going on??



Binary Search performance

```
SimpleTest BinarySearch
Tests from PROVIDED_TEST

Correct (PROVIDED_TEST, binsearch.cpp:88) Basic correctness: found value
Correct (PROVIDED_TEST, binsearch.cpp:93) Basic correctness: missing value
Correct (PROVIDED_TEST, binsearch.cpp:98) Edge case: found first value
Correct (PROVIDED_TEST, binsearch.cpp:103) Edge case: found last value
Correct (PROVIDED_TEST, binsearch.cpp:108) Timing on 10K elements
    Line 112 TIME_OPERATION binarySearch(data, 5) (size = 10000) completed in 0.000 secs
Correct (PROVIDED_TEST, binsearch.cpp:115) Timing on 100K elements
    Line 119 TIME_OPERATION binarySe
Correct (PROVIDED_TEST, binsearch.cpp:122)
    Line 126 TIME_OPERATION binarySe
Passed 7 of 7 tests. Great!
```

Q. We saw the test take a long time to run for 1M, but it reports 0.000 secs. What's going on??

Answer:
 $\log_2(10K) \approx 13$
 $\log_2(100K) \approx 16$
 $\log_2(1M) \approx 20$
...on a computer that does billions of operations per second!

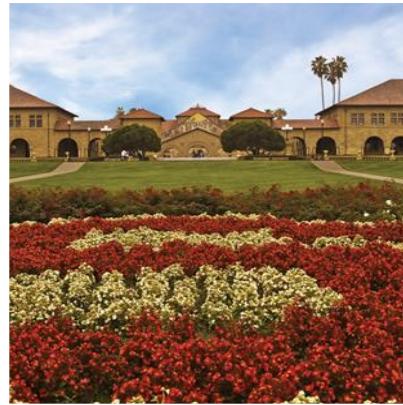
$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	1.84×10^{19}
7	128	896	16,384	3.40×10^{38}
8	256	2,048	65,536	1.16×10^{77}
9	512	4,608	262,144	1.34×10^{154}
10	1,024	9,216	1,048,576	1.80×10^{308}
11	2,048	18,432	(.0003s)	
12	4,096	36,864		
13	8,192	73,728		
14	16,384	147,456		
15	32,768	294,912		
16	65,536	589,824		
17	131,072	1,179,648		
18	262,144	2,359,296		
19	524,288	4,718,592		
20	1,048,576	9,437,184		
21	2,097,152	18,874,368		
22	4,194,304	37,748,736		
23	8,388,608	75,497,472		
24	16,777,216	150,994,944		
25	33,554,432	301,989,888		
26	67,108,864	603,979,776		
27	134,217,728	1,207,959,552		
28	268,435,456	2,415,918,104		
29	536,870,912	4,831,836,208		
30	1,073,741,824	9,663,672,416		
31	2,147,483,648	19,327,344,832		
32	4,294,967,296	38,654,688,664		
33	8,589,934,592	77,309,377,328		
34	17,179,869,184	154,618,754,656		
35	34,359,738,368	309,237,509,312		
36	68,719,476,736	618,475,018,624		
37	137,438,953,472	1,236,950,037,248		
38	274,877,906,944	2,473,900,074,496		
39	549,755,813,888	4,947,800,148,992		
40	1,099,511,627,776	9,895,600,297,984		
41	2,199,023,255,552	19,791,200,595,968		
42	4,398,046,511,104	39,582,400,191,936		
43	8,796,093,022,208	79,164,800,383,872		
44	17,592,186,044,416	158,329,600,767,744		
45	35,184,372,088,832	316,659,201,535,488		
46	70,368,744,177,664	633,318,403,070,976		
47	140,737,488,355,328	1,266,636,806,141,952		
48	281,474,976,710,656	2,533,273,612,283,904		
49	562,949,953,421,312	5,066,547,224,566,808		
50	1,125,899,906,842,624	10,133,094,449,133,616		
51	2,251,799,813,685,248	20,266,188,898,267,232		
52	4,503,599,627,370,496	40,532,377,796,534,464		
53	9,007,199,254,740,992	81,064,755,593,068,928		
54	18,014,398,509,481,984	162,129,511,186,137,856		
55	36,028,797,018,963,968	324,258,022,372,275,712		
56	72,057,594,037,927,936	648,516,044,744,551,424		
57	144,115,188,075,855,872	1,296,032,089,489,102,848		
58	288,230,376,151,711,744	2,592,064,178,978,205,696		
59	576,460,752,303,423,488	5,184,128,357,956,411,392		
60	1,152,921,504,606,846,976	10,368,256,715,912,822,784		
61	2,305,843,009,213,693,952	20,736,513,431,825,645,568		
62	4,611,686,018,427,387,904	41,473,026,863,651,291,136		
63	9,223,372,036,854,775,808	82,946,053,727,302,582,272		
64	18,446,744,073,709,551,616	165,892,107,454,605,164,544		
65	36,893,488,147,419,103,232	331,784,214,909,210,328,088		
66	73,786,976,294,838,206,464	663,568,429,818,420,656,176		
67	147,573,952,589,676,412,928	1,327,136,859,636,841,312,352		
68	295,147,905,179,352,825,856	2,654,273,719,273,682,624,704		
69	590,295,810,358,705,651,712	5,308,547,438,547,365,249,408		
70	1,180,591,620,717,411,303,424	10,617,094,877,094,730,498,816		
71	2,361,183,241,434,822,606,848	21,234,189,754,189,461,997,632		
72	4,722,366,482,869,644,413,696	42,468,379,508,378,923,995,264		
73	9,444,732,965,739,288,827,392	84,936,759,016,757,847,989,528		
74	18,889,465,931,478,575,654,784	169,873,518,033,515,695,978,056		
75	37,778,931,862,957,151,309,568	339,747,036,067,031,391,956,112		
76	75,557,863,725,914,302,618,136	679,494,072,134,062,783,912,224		
77	151,115,727,451,828,605,236,272	1,358,988,144,268,125,567,824,448		
78	302,231,454,903,657,210,472,544	2,717,976,288,536,251,135,648,896		
79	604,462,909,807,314,420,945,088	5,435,952,577,072,502,270,121,792		
80	1,208,925,819,614,628,841,890,176	10,871,905,154,145,004,540,243,584		
81	2,417,851,639,229,257,683,780,352	21,743,810,308,290,008,080,486,168		
82	4,835,703,278,458,515,367,560,704	43,487,620,616,580,016,160,972,336		
83	9,671,406,556,917,030,735,121,408	86,975,241,233,160,032,321,944,672		
84	19,342,813,113,834,061,470,242,816	173,950,482,466,320,064,643,889,344		
85	38,685,626,227,668,122,940,485,632	347,900,964,932,640,128,687,778,688		
86	77,371,252,455,336,244,880,971,264	695,801,928,865,280,256,375,557,376		
87	154,742,504,910,672,489,760,942,536	1,391,603,857,730,560,512,751,114,752		
88	309,485,009,821,344,979,521,885,072	2,783,207,715,461,120,025,502,229,504		
89	618,970,019,642,689,959,043,770,144	5,566,415,430,922,240,050,004,458,008		
90	1,237,940,039,285,379,918,087,540,288	11,132,830,861,844,480,025,008,916,176		
91	2,475,880,078,570,759,836,175,080,576	22,265,661,723,688,960,050,016,832,352		
92	4,951,760,157,141,519,672,350,160,152	44,531,323,447,377,920,100,032,664,704		
93	9,903,520,314,283,038,344,700,320,304	89,062,646,894,754,840,200,064,329,408		
94	19,807,040,628,566,076,689,400,640,608	178,125,293,789,508,680,400,128,658,816		
95	39,614,081,257,132,153,378,800,120,128	356,250,587,579,017,360,800,256,317,632		
96	79,228,162,514,264,306,757,600,240,256	712,501,175,158,034,721,600,512,635,264		
97	158,456,325,028,528,613,515,200,480,512	142,502,350,317,068,443,200,104,128,528		
98	316,912,650,057,057,227,030,400,960,024	285,004,700,634,136,886,400,208,256,056		
99	633,825,300,114,114,454,060,800,920,048	570,009,401,268,273,772,800,416,512,016		
100	1,267,650,600,228,228,908,120,160,840,096	1,140,018,802,536,547,545,600,832,032,032		
101	2,535,301,200,456,457,816,240,320,680,192	2,280,037,605,073,095,091,200,664,064,064		
102	5,070,602,400,912,914,632,480,640,360,384	4,560,075,210,146,190,182,400,136,128,128		
103	10,141,204,801,825,829,264,960,120,720,768	9,120,150,420,292,380,364,800,272,256,256		
104	20,282,409,603,651,658,529,920,240,140,156	18,240,300,840,584,760,729,600,544,512,512		
105	40,564,819,207,303,317,059,840,480,280,312	36,480,601,680,168,520,159,200,108,024,024		
106	81,129,638,414,606,634,119,680,960,560,624	72,961,203,360,336,040,318,400,216,048,048		
107	162,259,276,829,213,268,239,360,120,112,128	145,922,406,720,672,080,636,800,432,096,096		
108	324,518,553,658,426,536,478,720,240,224,256	291,844,813,440,144,160,133,600,864,192,192		
109	648,037,107,316,853,073,017,457,480,480,512	583,689,626,880,288,320,267,200,188,384,384		
110	1,296,074,214,633,706,146,034,914,960,960,024	1,167,379,253,760,576,640,534,400,376,768,768		
111	2,592,148,429,267,412,292,068,830,920,920,048	2,334,758,507,520,153,280,068,800,753,536,536		
112	5,184,296,858,534,824,584,136,661,860,860,096	4,669,517,015,040,306,560,136,800,156,712,712		
113	10,368,593,717,069,648,168,273,323,720,720,192	9,339,034,030,080,613,120,273,600,313,424,424		
114	20,737,187,434,139,296,336,546,647,440,440,384	18,678,068,060,160,126,240,546,800,626,848,848		
115	41,474,374,868,278,592,673,093,294,880,880,768	37,356,136,120,320,352,480,593,600,125,696,192		
116	82,948,749,736,557,185,346,186,588,760,760,156	74,712,272,240,640,704,960,593,200,251,392,384		
117	165,897,499,473,114,370,692,373,177,520,520,312	149,424,544,480,120,140,960,373,400,502,784,768		
118	331,794,998,946,228,740,784,746,355,040,040,624	298,849,088,960,240,280,960,746,800,754,568,536		
119	663,589,997,892,457,480,568,592,710,080,080,128	597,698,177,920,480,560,960,713,600,153,136,072		
120	1,327,179,995,784,914,960,137,185,420,160,160,256	1195,396,355,840,960,160,960,713,200,306,272,144		
121	2,654,359,991,569,829,920,274,370,840,320,320,512	2390,789,710,160,960,320,960,713,400,612,544,288		
122	5,308,719,983,139,659,858,548,741,680,640,640,024	47815,590,140,960,640,960,713,800,124,104,096		
123	10,617,439,966,278,319,717,097,483,360,120,120,048	95631,180,280,960,120,960,713,600,248,208,192		
124	21,234,879,932,556,638,434,194,967,720,240,240,096	191262,360,560,960,240,960,713,200,496,416,384		
125	42,469,759,865,113,277,868,388,935,440,480,480,192	382524,720,120,960,480,960,713,400,992,832,768		
126	84,939,519,730,226,554,736,776,870,880,960,960,384	765049,440,240,960,960,960,713,800,196,166,536		
127	169,879,039,460,453,108,473,553,741,760,192,192,768	153009,880,480,960,960,960,713,600,392,333,072		
128	339,758,078,920,906,216,947,106,483,520,384,384,536	306019,760,960,960,960,713,200,784,768,736,144		
129	679,516,157,841,812,432,994,212,967,040,768,768,072	612039,520,960,960,960,713,400,156,156,352,288		
130	1,358,988,315,683,644,865,424,934,420,153,536,016,144	122407,840,960,960,960,713,200,312,312,704,576		
131	2,717,976,631,367,289,730,848,848,848,784,784,032,288	244815,680,960,960,960,713,400,624,624,144,144		
132	5,435,953,262,734,578,461,672,176,176,156,156,064,576	489631,360,960,960,960,713,800,128,128,288,288		
133	10,871,906,525,469,156,943,344,352,352,312,312,128,128	979262,720,960,960,960,713,600,256,256,576,576		
134	21,743,813,050,938,312,986,688,704,704,728,728,256,256	195852,440,960,960,960,713,200,512,512,112,112		
135	43,487,620,101,876,624,973,376,408,408,456,456,512,512	391704,880,960,960,960,713,400,102,102,224,224		
136	86,975,240,203,753,248,946,752,816,816,816,928,928,024	783408,160,960,960,960,713,200,204,204,448,448		
137	173,950,480,407,506,496,993,504,163,163,163,185,185,048	156681,680,960,960,960,713,400,408,408,896,896		
138	347,900,960,814,012,992,992,992,371,371,371,371,371,096	313362,360,960,960,960,713,800,816,816,178,178		
139	695,801,920,628,024,984,984,984,742,742,742,742,742,192	626724,720,960,960,960,713,600,163,163,356,356		
140	1,391,603			

Big-O Key Take-Aways:

- **NOT worth doing:** Optimization of your code that **just trims** a bit
 - › Like that $+/-1$ handshake—we don't need to worry ourselves about it!
 - › **Just write clean, easy-to-read code!!!!**
- **MAY be worth doing:** Optimization of your code that **changes Big-O**
 - › If performance of a particular function is important, focus on this!
 - › *(but if performance of the function is not very important, for example it will only run on small inputs, focus on just writing clean, easy-to-read code!!)*
- (Also remember that efficiency is not necessarily a virtue—first and foremost focus on correctness, both technical and ethical/moral/societal justice)

Heads or Tails?

GENERATING SEQUENCES



Heads or Tails?

- You flip a coin 5 times
- What are all the possible heads/tails sequences you could observe?
 - › TTTTT
 - › HHHHH
 - › THTHT
 - › HHHHT
 - › etc...
- We want to write a program to fill a Vector with strings representing each of the possible sequences.



Generating all possible coin flip sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```

Your Turn: coin flip sequences



```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```

- **Q: Of these sequences (all of which should be included in allSequences), which sequence appears first in allSequences? Last?**
 - TTTTT, HHHHH, THTHT, HHHHT

Your Turn: coin flip sequences



```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```

- **Q: What would happen if we didn't do the erase (highlighted above)? Which of the following sequences would we NOT generate? Which additional sequences would we generate (that we shouldn't)?**
 - TTTTT, HHHHH, THTHT, HHHHT