SCC.121: ALGORITHMS AND COMPLEXITY
Sentinel and Binary Search Algorithms

Emma Wilson e.d.wilson1@lancaster.ac.uk

Today’s Lecture

Aim: Look at searching algorithms sentinel search and binary search and their time complexity. Introduce asymptotic analysis and the growth of functions.

Learning objectives:

- Know how sentinel and binary search work and be able to estimate their time complexity
- To know what is meant by the growth rate of functions and be able to determine the order of growth of simple functions

Outline

- Linear Search vs Sentinel Search
- Binary Search
- Introduction to growth of functions

Outline

- **Linear Search vs Sentinel Search**
- Binary Search
- Introduction to growth of functions

Linear Search

Lancaster University

The overall program time (Worst case)?

- **T(N) = 3N+3**

```
int isInArray(int theArray[], int N, int iSearch)
{

               o1 → 1  o2 → N+1  o3 → N
       for (int i = 0; i < N; i++)
                                  o4 → N
              if (theArray[i] == iSearch)

                     return 1;      o5 → 0

       return 0;      o6 → 1
}
```

Sentinel Search

- When a linear search is performed on an array of size N then in the worst case a total of (N + 1) comparisons are made for the index of the element to be compared so that the **index is not out of bounds of the array** .
- Sentinel search is a type of Linear search where **the number of comparisons is reduced** as compared to the linear search.

Sentinel Search

Lancaster University

```
int isInSentinel(int theArray[], int N,  int iSearch)
{
        // Sentinel Search code
}
```

	0	1	2	3	4	5	6	7	8	9
theArray	5	17	6	13	28	1	3	6	44	32

iSearch 6 The number we are looking for in the array

- ❑ Check if `iSearch` is not at the end of `theArray`. Then, replace the number at the end of array with `iSearch`

	0	1	2	3	4	5	6	7	8	9
theArray	5	17	6	13	28	1	3	6	44	6
										iSearch

- ❑ Perform Linear Search

Sentinel Search

```
int isInSentinel(int theArray[], int N, int iSearch)
{

        if (theArray[N-1] == iSearch)
                return 1;

        theArray[N-1] = iSearch;

        for (int i=0;      ; i++) {

                if (theArray[i] == iSearch)
                        return (i < (N-1));
        }
}
```

slido

How many times do we execute i++ in sentinel search in the worst case?

ⓘ Start presenting to display the poll results on this slide.

Sentinel Search (worst case)

```
int isInSentinel(int theArray[], int N, int iSearch)
{

        if (theArray[N-1] == iSearch) o1 → 1
                return 1; o2 → 0

        theArray[N-1] = iSearch; o3 → 1
            o4 → 1            o4 → N-1
        for (int i=0;[    ]; i++) {

                if (theArray[i] == iSearch) o4 → N
                        return (i < (N-1)); o5 → 1
        }
}
```

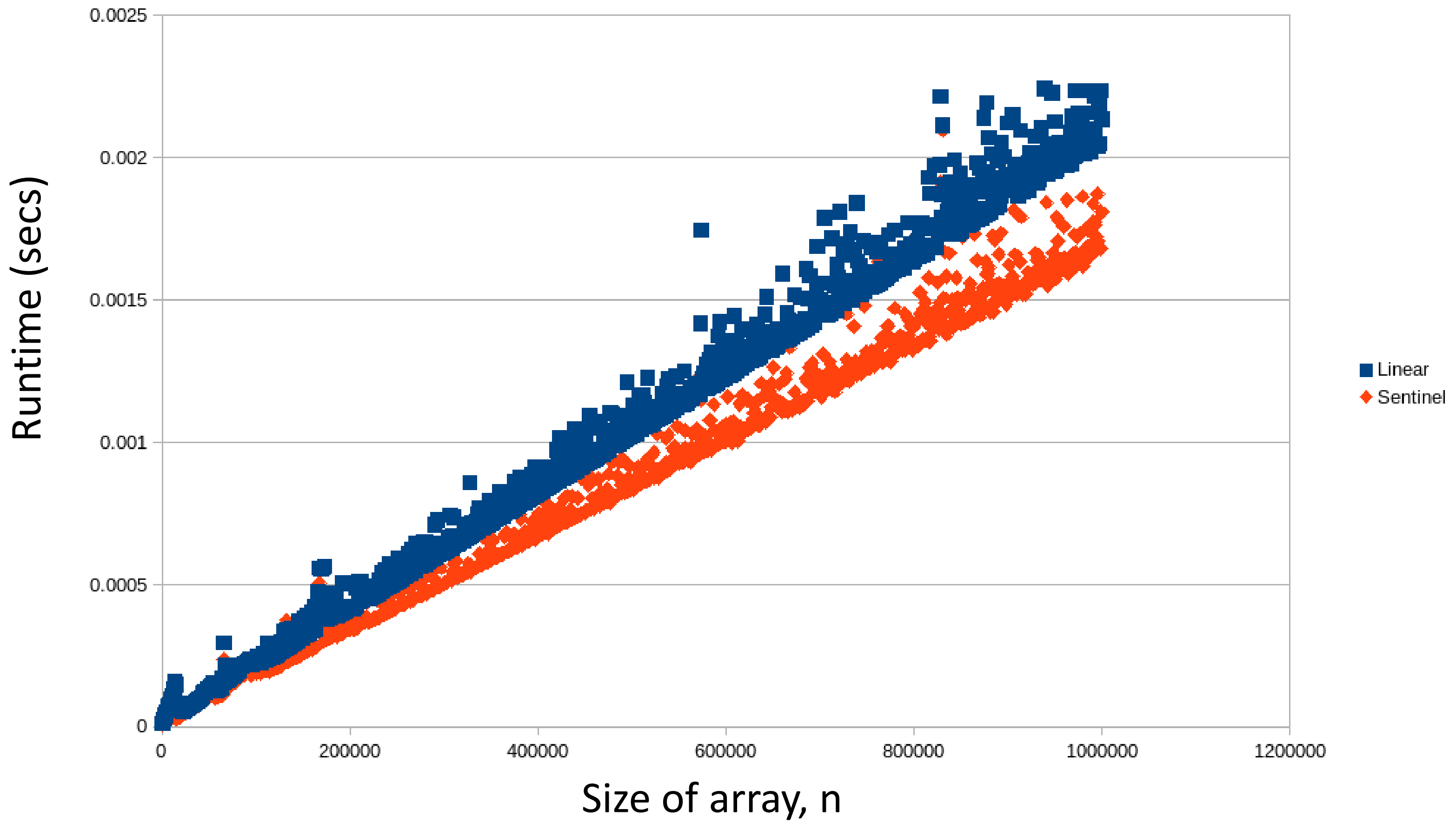
Compare Linear vs. Sentinel Search

```
int isInArray(int theArray[], int N,
int iSearch)
{
for (int i=0; i<N; i++)
        if (theArray[i] == iSearch)
            return 1;
    return 0;
}
```

```
int isInSentinel(int theArray[], int N,
int iSearch)
{
        if (theArray[N-1] == iSearch)
         return 1;
    theArray[N-1] = iSearch;
    for (int i=0;     ; i++)
    {
        if (theArray[i] == iSearch)
            return (i < (N-1));
    }
}
```

- Which one is better?
  - Count the number of instructions
    - Linear: **3N+3** (Worst-case)
    - Sentinel: **2N+3** (Worst-case)
  - Both searches have **linear** time complexity

Linear vs Sentinel Search: Actual Runtimes (worst case)



Outline

- Linear Search vs Sentinel Search
- **Binary Search**
- Growth of functions

Binary Search

- Binary search works on **sorted arrays**
- Locates a target value in a sorted array by successively eliminating half of the array from consideration
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo							↑ mid							↑ hi

$$\boldsymbol{mid} = \frac{(\boldsymbol{hi} + \boldsymbol{lo})}{\mathbf{2}}$$

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating <u>half</u> of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo							↑ mid							↑ hi

$$mid = \frac{(hi + lo)}{2}$$

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let’s assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo						↑ hi								

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo			↑ mid			↑ hi								

$$mid = \frac{(hi + lo)}{2}$$

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				lo		hi								

Binary Search

Lancaster University

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo	↑ mid	↑ hi								

$$mid = \frac{(hi + lo)}{2}$$

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑

lo
hi

Binary Search

- Locates a target value in a <u>*sorted*</u> array by successively eliminating <u>half</u> of the array from consideration.
- Let’s assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

lo
hi
mid

Binary Search

- Locates a target value in a ***sorted*** array by successively eliminating half of the array from consideration.
- Let's assume your given the below array and $iSearch = 33$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑

lo
hi
mid

Binary Search Algorithm (an iterative implementation)

```
boolean isInBinary(int[] theArray, int N, int iSearch)
{
	int lo = 0;
	int hi = N - 1;
	int mid = 0;

	while (hi >= lo) {
		mid = (lo + hi)/2; //round to higher integer
		if (theArray[mid] == iSearch)
			return true;
		else if (theArray[mid] < iSearch)
			lo = mid + 1;
		else
			hi = mid - 1;
	}
	return false;
}
```

Binary Search Questions

- What is the time complexity in the Worst-case?
- What is the time complexity in the Best-case?

Binary Search Algorithm (an iterative implementation)

Lancaster University

```
boolean isInBinary(int[] theArray, int N, int iSearch)
{
     int lo = 0;   o1
     int hi = N - 1;  o2
     int mid = 0;  o3

     while (hi >= lo) {  o4
          mid = (lo + hi)/2; //round to higher integer  o5
          if (theArray[mid] == iSearch)  o6
               return true;  o7
          else if (theArray[mid] < iSearch)  o8
               lo = mid + 1;  o9
          else
               hi = mid - 1;  o10
     }
     return false;  o11
}
```

Worst Case:

- iSearch is not in the array
- o6 always false
- Assume o8 always false

Binary Search (Worst Case)

Try out for an array size: here try $N = 2^6 = 64$

	lo	hi	mid
Initially	0	63	0
After 1st search	0	31	32
After 2nd search	0	15	16
After 3rd search	0	7	8
After 4th search	0	3	4
After 5th search	0	1	2
After 6th search	0	0	1
After 7th search	0	-1	0

- Here $N = 2^6 = 64$
- Or equivalently $7 = \log_2 64 + 1 = \log_2 N$+1
- So, we go around the loop $\log_2 N + 1$ times
- Each time around the loop o4, o5,o6,o8 and o10 are implemented
- o4 is implemented one additional time
- o1, o2, o3 and o11 are implemented once

Total time complexity in worst case

- T(N) = o1+o2+o3+o4+o5+o6+o8+o10+o11
- T(N) = 1+1+1+5*(1+ $\log_2 N) + 1$+1
- T(N) = 10+5 $\log_2 N$

Worst Case time complexity is logarithmic! $T(N) = \mathbf{C_1} + \mathbf{C_2 logN}$

Binary Search Algorithm

```
boolean isInBinary(int[] theArray, int N, int iSearch)
{
     int lo = 0;
     int hi = N - 1;
     int mid = 0;

     while (hi >= lo) {
          mid = (lo + hi)/2; //round to higher integer
          if (theArray[mid] == iSearch)
               return true;
          else if (theArray[mid] < iSearch)
               lo = mid + 1;
          else
               hi = mid - 1;
     }
     return false;
}
```

slido

What is the time complexity class of Binary Search Algorithm in the best case?

ⓘ Start presenting to display the poll results on this slide.

Binary Search Questions

- What is the time complexity in the Worst-case?
  - **Logarithmic:** $T(N) = \mathbf{C_1 + C_2 logN}$
- What is the time complexity in the Best-case?
  - **Constant:** $T(N) = \mathbf{C_1}$

Linear, Sentinel and Binary Search Summary

- Sentinel search and Linear search algorithms are both linear in worst-case scenario, but Sentinel search requires executing fewer operations
- Binary search algorithm is more efficient comparing to Linear and Sentinel Search algorithms (Binary logarithmic in worst case), but the input array should be sorted
- Actual values of constants generally unimportant (except in specific circumstances)
- What we really care about is behaviour as the size of our input increases – **asymptotic behaviour**

Outline

- Linear Search vs Sentinel Search
- Binary Search
- **Introduction to growth of functions**

The Growth of Functions

- For example, let us assume two algorithms A and B that solve the same class of problems
- The time complexity of **A** is $T(n) = 5000n$, the one for **B** is $T(n) = 1.1^n$ for an input with n elements
- Which algorithm is better?

The Growth of Functions

Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	T(n) = 5,000n	$T(n)=1.1^n$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5\text{x}10^{41}$
1,000,000	$5\text{x}10^9$	$4.8\text{x}10^{41398}$

slido

The time complexity of A is $T(n) = 5000n$, the one for B is $T(n) = 1.1^n$ for an input with n elements, Which algorithm is better?

ⓘ Start presenting to display the poll results on this slide.

The Growth of Functions

- This means that algorithm B <u>cannot</u> be used for large inputs, while algorithm A is still feasible

- So what is important is the **growth** of the time complexity functions

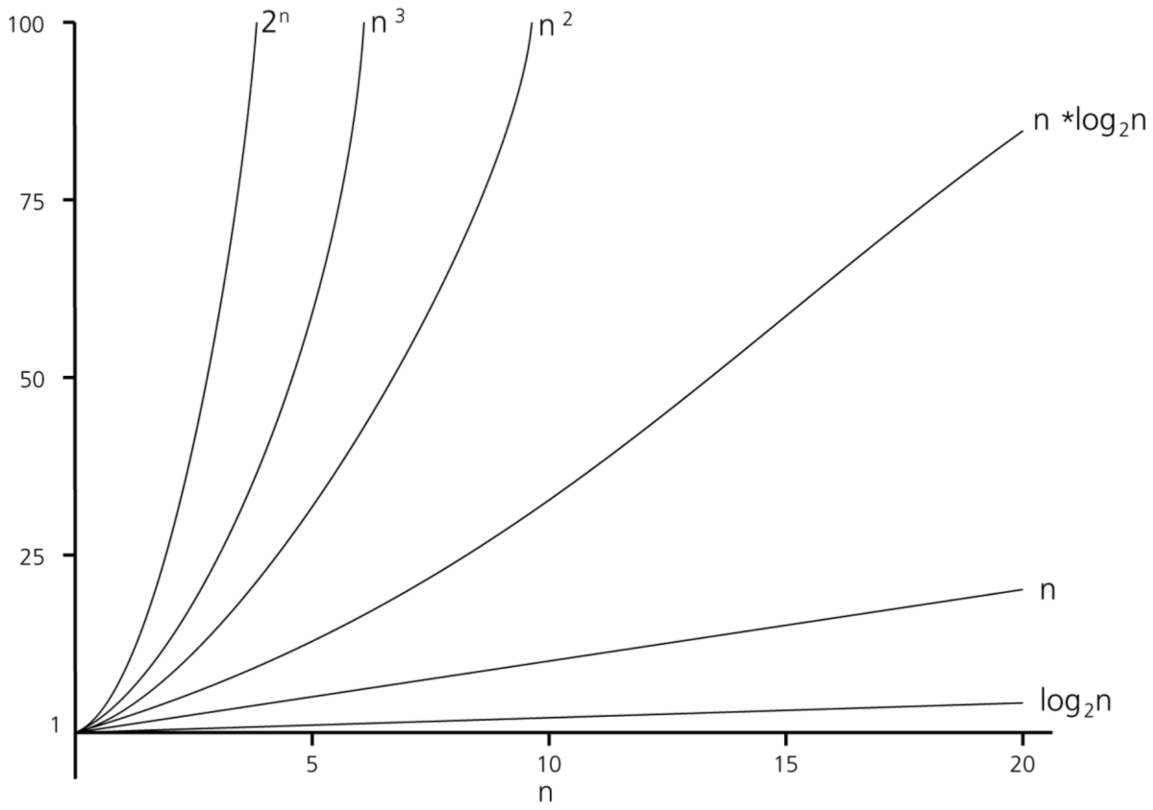
- The growth of time complexity with increasing input size ‘n’ is a suitable measure for the comparison of algorithms

The Growth of Functions (Table)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Order of Growth

The Growth of Functions (Plot)



The Growth of Functions (Summary)

c is a constant $0 < c < 1$

$$\log n \prec n^c \prec n \prec n \log n \prec n^2 \prec n^3 \ldots \prec 2^n \prec 3^n \prec \cdots \prec n!$$

For example: $\sqrt{n} = n^{\frac{1}{2}}$

Growth rate of functions

Listed from slowest to fastest growth:

- **1** → Constant growth
- **log n** → Logarithmic growth
- **n^c** → where 0<c<1
- **n** → Linear growth
- **n log n**
- **n^2** → Quadratic growth
- **n^2 log n**
- **n^3** → Cubic growth
- **n^c** → Polynomial growth (c is a constant number)
- **2^n** → Exponential growth
- **3^n** → Exponential growth
- **c^n** → Exponential growth (c is a constant number)
- **n!** → Factorial growth

Constant$\prec \boldsymbol{log\, n} \prec \boldsymbol{n^c}$ (0<c<1)$\prec \boldsymbol{n} \prec \boldsymbol{n\, log\, n} \prec \boldsymbol{n^2} \prec \boldsymbol{n^3} \ldots \prec \boldsymbol{2^n} \prec \boldsymbol{3^n} \prec \cdots \prec \boldsymbol{n}!$

- $T_1(n) = (1.5)^n$
- $T_2(n) = 8n^3 + 17\, n^2 + 11$
- $T_3(n) = \log(n)$
- $T_4(n) = 2^n$
- $T_5(n) = \log(\log(n))$
- $T_6(n) = n^2 \log(n)$
- $T_7(n) = 2^n(n^2 + 1)$
- $T_8(n) = 100000$
- $T_9(n) = n!$

- $T_8(n) = 100000$
- $T_5(n) = \log(\log(n))$
- $T_3(n) = \log(n)$
- $T_6(n) = n^2 \log(n)$
- $T_2(n) = 8n^3 + 17\, n^2 + 11$
- $T_1(n) = (1.5)^n$
- $T_4(n) = 2^n$
- $T_7(n) = 2^n(n^2 + 1)$
- $T_9(n) = n!$

Please download and install the Slido app on all computers you use

Audience Q&A

ⓘ Start presenting to display the audience questions on this slide.

Summary

Today's lecture: focused on sentinel and binary search algorithms and their time complexity. Introduced the growth of functions

- Sentinel search and Linear search algorithms are both linear in worst-case scenario, but Sentinel search requires executing fewer operations
- Binary search algorithm is more efficient comparing to Linear and Sentinel Search algorithms, but the input array should be sorted
- In most cases - the growth of time complexity with increasing input size 'n' is a suitable measure for the comparison of algorithms. Useful to know the order in which simple functions grow!

Next Lecture: Asymptotic analysis and Big-O notation