Algorithm Analysis

Outline

- Run time analysis
- Growth rate of functions
- Big-O notation
- Examples
- Worst-case, best-case and average case analysis of algorithms

Algorithm

- An ***algorithm*** is a set of instructions to be followed to solve a problem.
  - There can be more than one solution (more than one algorithm) to solve a given problem.
  - An algorithm can be implemented using different programming languages on different platforms.
- An algorithm must be correct. It should correctly solve the problem.
  - e.g. For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

Algorithmic Performance

There are *two aspects* of algorithmic performance:

- Time
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?
- Space
  - Data structures take space
  - What kind of data structures can be used?
  - How does choice of data structure affect the runtime?

➢ We will focus on time:

- How to estimate the time required for an algorithm
- How to reduce the time required

Analysis of Algorithms

- When we analyze algorithms, we employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

The Running Time of Algorithms

- Each operation in an algorithm (or a program) has a cost.

  ➔ Each operation takes a certain of time.

`count = count + 1;` ➔ take a certain amount of time, but it is constant

A sequence of operations:

`count = count + 1;`	Cost: c_1
`sum = sum + count;`	Cost: c_2

➔ Total Cost = $c_1 + c_2$

Run Time Analysis

Example: *Simple If-Statement*

	Cost	**Times**
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

Run Time Analysis (cont.)

Example: *Simple Loop*

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

Exercise

Which of these loops takes constant time regardless of the value of n?

(a) for (i=n/10; i<n; i++) sum+=i;

(b) for (i=0; i<n; i+=n/10) sum++;

(c) for (i=n; i<2*n; i++) sum--;

(d) for (i=0; i<n; i+=10) sum++;

(e) for (i=0; i<n/10; i+=10) sum*=2;

Run Time Analysis (cont.)

Example: Nested Loop

Code	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to $\mathbf{n^2}$

Example: Nested Loop

Problem: given n numbers in an array A, calculate the sum of all **distinct** pairwise multiplications.

```
// A is an array of integers of size n
double sum = 0.0;
for (int i=0; i < n; i++) {
    for (int j=i; j < n; j++) {
        sum += A[i]*A[j];
    }
}
return sum;
```

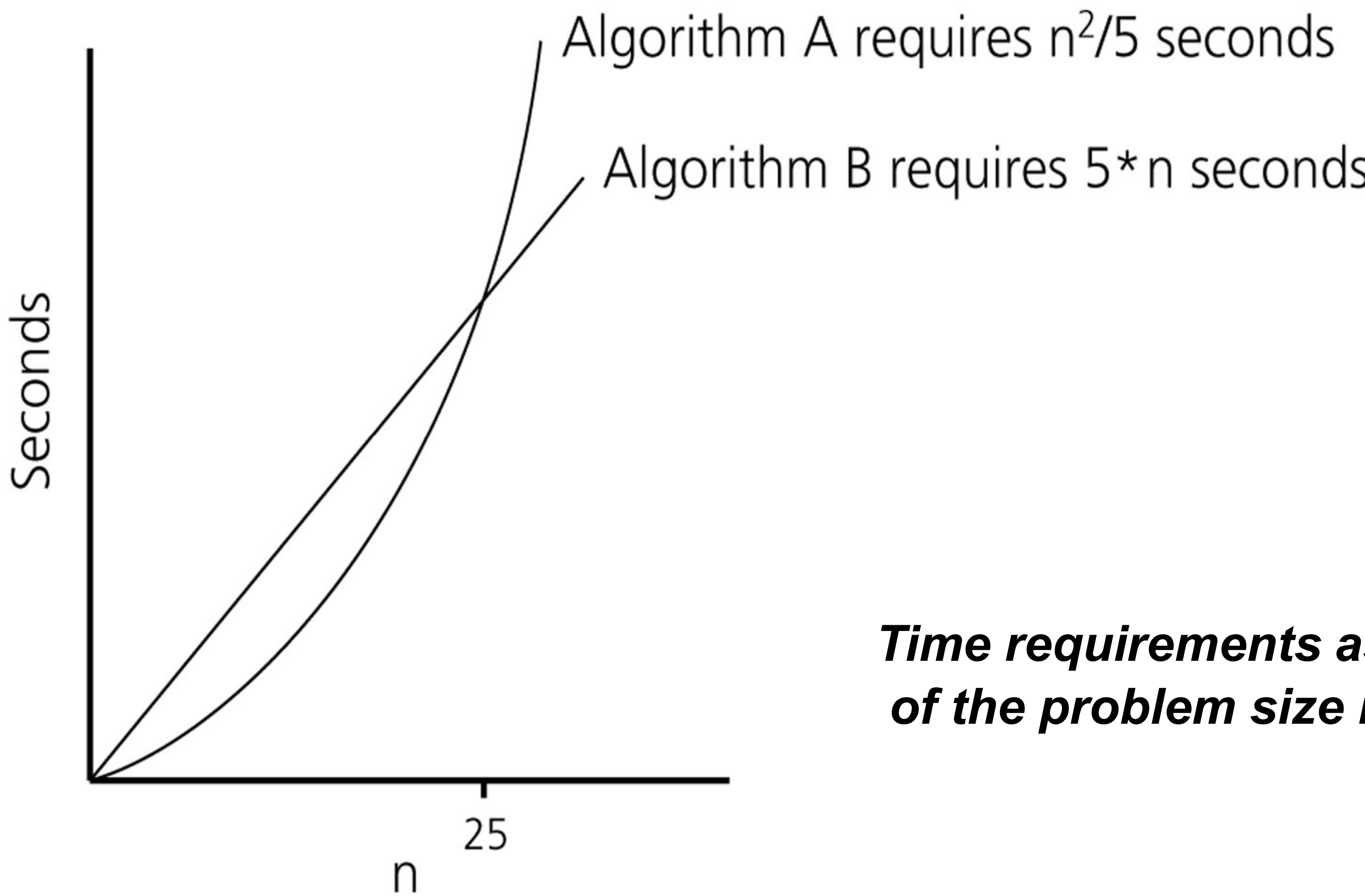
$$n + (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n+1)}{2}$$

➔ The time required for this algorithm is proportional to $\mathbf{n^2}$

Algorithm Growth Rates

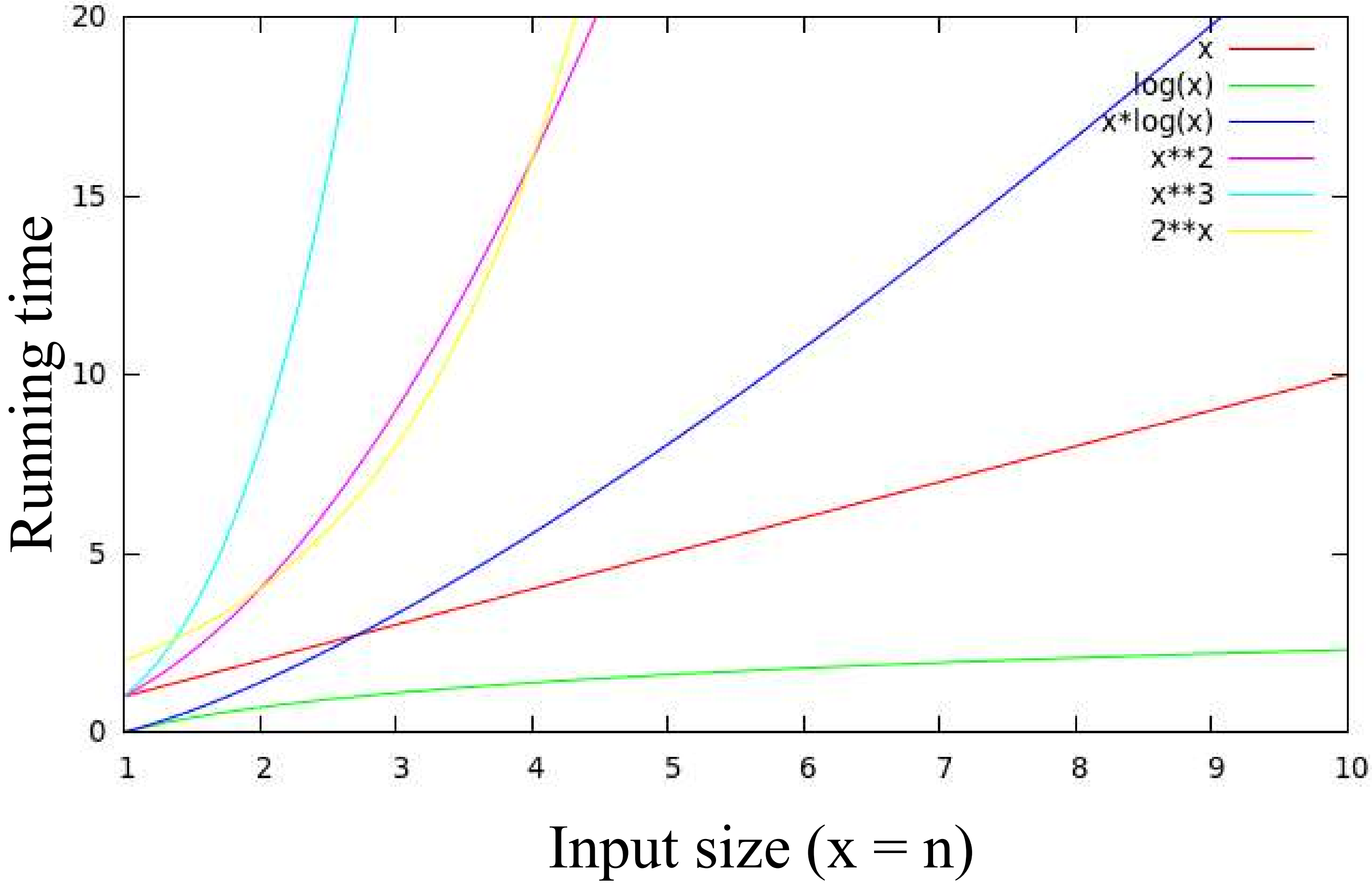
- We measure an algorithm's time requirement as a function of the *problem size*.
  - Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number disks for towers of hanoi.
- The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.
  - Algorithm A requires time proportional to $\mathbf{n^2}$.
  - Algorithm B requires time proportional to **n**.
- An algorithm's proportional time requirement is known as ***growth rate***.
- We can compare the efficiency of two algorithms by comparing their growth rates.

Algorithm Growth Rates (cont.)

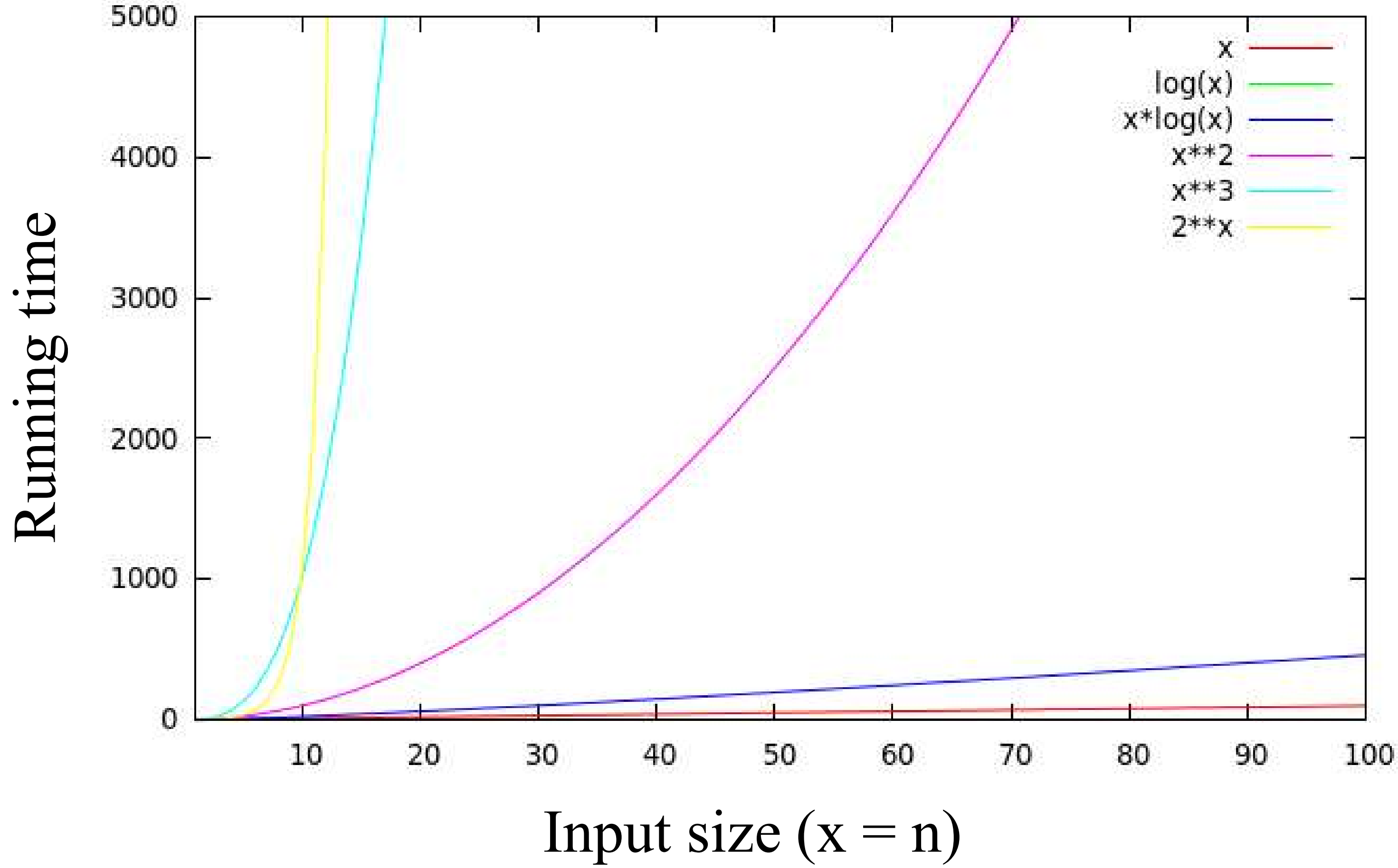


Time requirements as a function of the problem size n

Running Times for Small Inputs



Running Times for Large Inputs



Big-O Notation

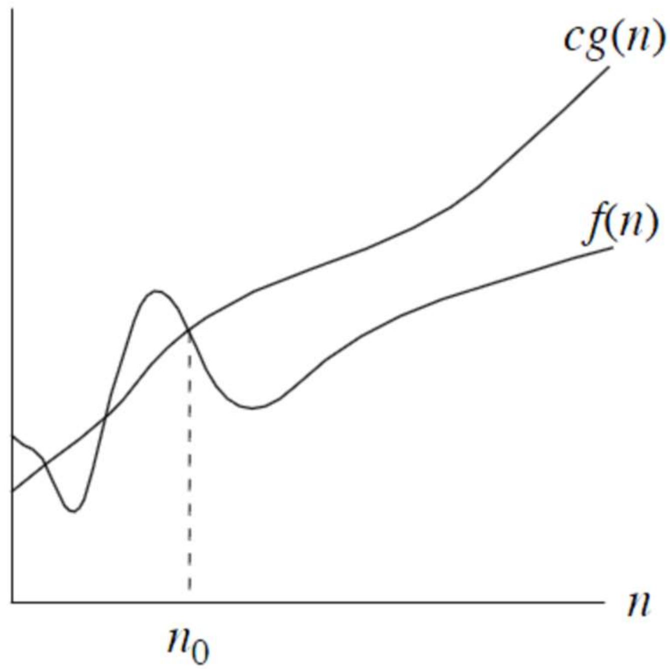
- **Big O notation** is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

- We use Big O notation to describe the computation time (**complexity)** of algorithms using algebraic terms.

- O stands for ‘**order**’, as in ‘order of magnitude’.

O – Notation (Formally)

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

Big-O Example

- If an algorithm requires $2n^2–3n+10$ seconds to solve a problem size n and constants c and n_0 exist such that

  $$2n^2–3n+10 \leq cn^2 \quad \text{for all } n \geq n_0 .$$

- In fact, for c = 3 and n_0 = 3:

  $$2n^2–3n+10 \leq 3n^2 \quad \text{for all } n \geq 3 .$$

- Thus, we say that the algorithm requires no more than $3n^2$ steps for $n \geq 3$, so it is **$O(n^2)$**
  - The fastest growing term is $2n^2$.
  - The constant 2 can be ignored

Order of Terms

- If we graph $0.0001n^2$ against $10000n$, the linear term would be larger for a long time, but the quadratic one would eventually catch up (here at $n = 10^8$).

- In calculus we know that

- $\lim_{n\to\infty} \frac{10000n}{0.0001n^2} = \lim_{n\to\infty} \frac{10^8}{n} = 0$

- As you can see, any quadratic (with a positive leading coefficient) will eventually beat any linear. So the linear term in a quadratic function eventually does not matter.

Order of Terms

- Consider the function $\mathbf{n^4 + 100\ n^2 + 500 = O(n^4)}$

n	n^4	$100n^2$	500	f(n)
1	1	100	500	601
10	10,000	10,000	500	20,500
100	100,000,000	1,000,000	500	101,000,500
1000	1,000,000,000,000	100,000,000	500	1,000,100,000,500

- The growth of a polynomial in n, as n increases, depends primarily on the degree (the highest order term), and not on the leading constant or the low-order terms

Big-O Summary

- Write down the cost function (i.e. number of instructions in terms of the problem size n)
  - Specifically, focus on the loops and find out how many iterations the loops run
- Find the highest order term
- Ignore the constant scaling factor.
- Now you have a Big-O notation.

Exercise 1

What is the complexity in Big-O notation?

Code Fragment

```
i=1;
while (i<n) {
    j = 1;
    while (j<100) {
        j=j+1;
    }
    i=i+1;
}
```

(a) $O(n)$
(b) $O(n^2)$
(c) $O(\log_2 n)$
(d) $O(2^n)$
(e) $O(n/2)$

Exercise 2

What is the complexity in Big-O notation?

Code Fragment

`int count = 0;`
`for (int i =1; i<n; i*=2)`
`     count++;`

(a) $O(n)$
(b) $O(n^2)$
(c) $O(\log_2 n)$
(d) $O(2^n)$
(e) $O(n/2)$

Logarithmic Cost O(log n)

```
for(int i=1; i < n; i*=2) {…}

for(int i=1; i < n; i<<=1) {…}
```

base 2

```
for(int i=n; i>0; i/=3) {…}
```

base 3

```
for(int i=n; i>0; i>>=2) {…}
```

base 4

The base does not matter, because:

$$O(\log_2 n) = O(\ln n)/O(\ln 2) = O(\ln n)$$

Change of base -> Base e (natural log)

Common Growth Rates

Function	Growth Rate Name
c	Constant
$log\ N$	Logarithmic
$log^2 N$	Log-squared
N	Linear
$N\ log\ N$	Log-linear (Linearithmic)
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Growth-Rate Functions

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
- If its order is:

$O(1)$ ➔ $T(n) = 1$ second

$O(\log_2 n)$ ➔ $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds

$O(n)$ ➔ $T(n) = (1*16) / 8 = 2$ seconds

$O(n*\log_2 n)$ ➔ $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds

$O(n^2)$ ➔ $T(n) = (1*16^2) / 8^2 = 4$ seconds

$O(n^3)$ ➔ $T(n) = (1*16^3) / 8^3 = 8$ seconds

$O(2^n)$ ➔ $T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds

Exercise 3

What is the complexity in Big-O notation?

Code Fragment
i=1; **while (i<N*N)** **i = i+(N/2);**

Exercise 4

What is the complexity in Big-O notation?

Code Fragment

`i=1;`
`while (i<N*N)`
`  i = 2*i;`

Exercise 5

What is the complexity in Big-O notation?

Code Fragment

```
 i=1;
while (i<N) {
   j = 1;
   while (j<N)
      j=j+1;
   i=2*i;
}
```

Exercise 6 (hard)

What's the complexity in Big-O notation?

Code Fragment

`int i, j, count=0;`
`for (i=1; i < N; i*=2) {`
`  for (j=0; j < i; j++) {`
`    count ++;`
`  }`
`}`

Exercise 7

What is the complexity in Big-O notation?

Code Fragment

```
x = 1;
for (i = 0; i <= N-1; i++) {
    for (j = 1; j <= x; j++)
      cout << j << endl;
    x = x * 2;
}
```

Outline

- Run time analysis
- Growth rate of functions
- Big-O notation
- Examples
- Worst-case, best-case and average case analysis of algorithms

What to Analyze

- An algorithm can require different times to solve different problems of the same size.
- Eg. Searching an item in an array of n elements using sequential search.
- ➔ Cost : 1, 2, 3, … n

What to Analyze

- ***Worst-Case Analysis*** –The maximum amount of time that an algorithm require to solve a problem of size n.
  - This gives an upper bound for the time complexity of an algorithm.
  - Normally, we try to find worst-case behavior of an algorithm.

- ***Best-Case Analysis*** –The minimum amount of time that an algorithm require to solve a problem of size n.
  - The best case behavior of an algorithm is NOT so useful.

- ***Average-Case Analysis*** –The average amount of time that an algorithm require to solve a problem of size n.
  - Sometimes, it is difficult to find the average-case behavior of an algorithm.
  - We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.
  - ***Worst-case analysis is more common than average-case analysis.***

Sequential Search

```
int sequentialSearch(const int a[], int item, int n)
{
   for (int i = 0; i < n && a[i]!= item; i++);

   if (i == n)
        return -1;
   return i;
}
```

Unsuccessful Search: $\rightarrow$ O(n)

Successful Search:

Best-Case: *item* is in the first location of the array $\rightarrow$O(1)

Worst-Case: *item* is in the last location of the array $\rightarrow$O(n)

Average-Case: The number of key comparisons 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2 + n)/2}{n}$$ $\rightarrow$ O(n)

Binary Search

```
int binarySearch(int a[], int size, int x)
{
   int low =0;
   int high = size -1;
   int mid;             // mid will be the index of
                        // target when it's found.
   while (low <= high) {
     mid = (low + high)/2;
     if (a[mid] < x)
        low = mid + 1;
     else if (a[mid] > x)
         high  = mid - 1;
     else
         return mid;
   }
   return -1;
}
```

Binary Search – Analysis

- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$

    ➔ $O(\log_2 n)$
- For a successful search:
  - ***Best-Case:*** The number of iterations is 1. ➔ $O(1)$
  - ***Worst-Case:*** The number of iterations is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$
  - ***Average-Case:*** The avg. # of iterations < $\log_2 n$ ➔ $O(\log_2 n)$

0	1	2	3	4	5	6	7	← an array with size 8
3	2	3	1	3	2	3	4	← # of iterations

The average # of iterations = $21/8 < \log_2 8$

How much better is $O(\log_2 n)$?

n	$O(\log_2 n)$
16	4
64	6
256	8
1024 (1KB)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1MB)	20
1,073,741,824 (1GB)	30

Analysis of Recursive Functions

Recursive functions

```
int naivePower(int x, int n){
   if (n == 0)
      return 1;
   else
      return (x * naivePower(x, n – 1));
```

How can we write the running time?

=> Recurrence relations

Recurrence Relation

- A recurrence relation is an equation that recursively defines a function's values in terms of earlier values
- Very useful for analyzing an algorithm's running time!

Recurrence relations for naïvepower code

$T(0) = c_1$

$T(n) = c_2 + T(n - 1)$

If only we had an expression for $T(n - 1)$…

Recurrence relations for code

$T(0) = c_1$

$T(n) = c_2 + T(n - 1)$

Expanded:

$T(n) = c_2 + (c_2 + T(n - 2))$

$= c_2 + (c_2 + (c_2 + T(n - 3)))$

$= c_2 + (c_2 + (c_2 + (c_2 + T(n - 4))))$

…

$= k\,c_2 + T(n - k)$

After n expansions: $n\,c_2 + T(0) = n\,c_2 + c_1$

=> $O(n)$

Another Example

```
int betterPower(int x, int n):
   if (n == 0)
      return 1;
   else if (n == 1)
      return x;
   else
      return betterPower(x * x, n/2);
```

(assume for simplicity that n is a power of 2)

How can we write the running time?

Recurrence relation for code

$T(0) = c_1$

$T(1) = c_2$

$T(n) = c_3 + T(n/2)$

(for simplification we'll assume n is a power of 2)

If only we had an expression for $T(n/2)$…

Recurrence relation for code

$T(0) = c_{1,}\ \ T(1) = c_2,\ \ \ldots, T(n) = c_3 + T(n/2)$

Expanded:

Recurrence relation for code

$T(0) = c_{1,}\ \ T(1) = c_2,\ \ \ldots, T(n) = c_3 + T(n/2)$

Expanded:

$$
\begin{aligned}
T(n) &= c_3 + T(n/2) \\
&= c_3 + c_3 + T(n/4) \\
&= c_3 + c_3 + c_3 + T(n/8)
\end{aligned}
$$

…..

$$= kc_3 + T(n/(2^k))$$

What should k be in order for us to get down to T(1)?

$2^k = n$, so $k = \log n$

$$\boxed{= c_3 * \log n}$$

=> O(log n)

Hanoi Towers Problem

```
void hanoi(int n, char source, char dest, char spare) {      Cost
  if (n > 0) {                                                c1
    hanoi(n-1, source, spare, dest);                          c2
     cout << "Move top disk from pole " << source             c3
          << " to pole " << dest << endl;
    hanoi(n-1, spare, dest, source);                          c4
} }
```

Hanoi Towers

- What is the cost of `hanoi(n,'A','B','C')`?

when n=0

$T(0) = c1$

when n>0

$$
\begin{aligned}
T(n) &= c1 + c2 + T(n-1) + c3 + c4 + T(n-1) \\
&= 2*T(n-1) + (c1+c2+c3+c4) \\
&= \mathbf{2*T(n-1) + c}
\end{aligned}
$$

$\Leftarrow$ recurrence equation for the growth-rate function of hanoi-towers algorithm

- Now, we have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm

Hanoi Towers (cont.)

$T(n) = 2*T(n-1) + c$

$= 2 * (2*T(n-2)+c) + c$

$= 2 * (2* (2*T(n-3)+c) + c) + c$

$= 2^3 * T(n-3) + (2^2+2^1+2^0)*c$ (assuming n>2)

when substitution repeated i-1th times

$= 2^i * T(n-i) + (2^{i-1}+ \ldots +2^1+2^0)*c$

when i=n

$= 2^n * T(0) + (2^{n-1}+ \ldots +2^1+2^0)*c$

$= 2^n * c1 + (\sum_{i=0}^{n-1} 2^i)*c$

$= 2^n * c1 + (2^n-1)*c = 2^n*(c1+c) - c$ ➔ So, the growth rate function is **$O(2^n)$**

Exercise

- What is the running time of recursive Fibbonacci function?

```
int Fib(int n){
   if (n == 1) return 1;
   if (n == 2) return 1;
   return Fib(n-2) + Fib(n-1);
}
```

Linked Lists

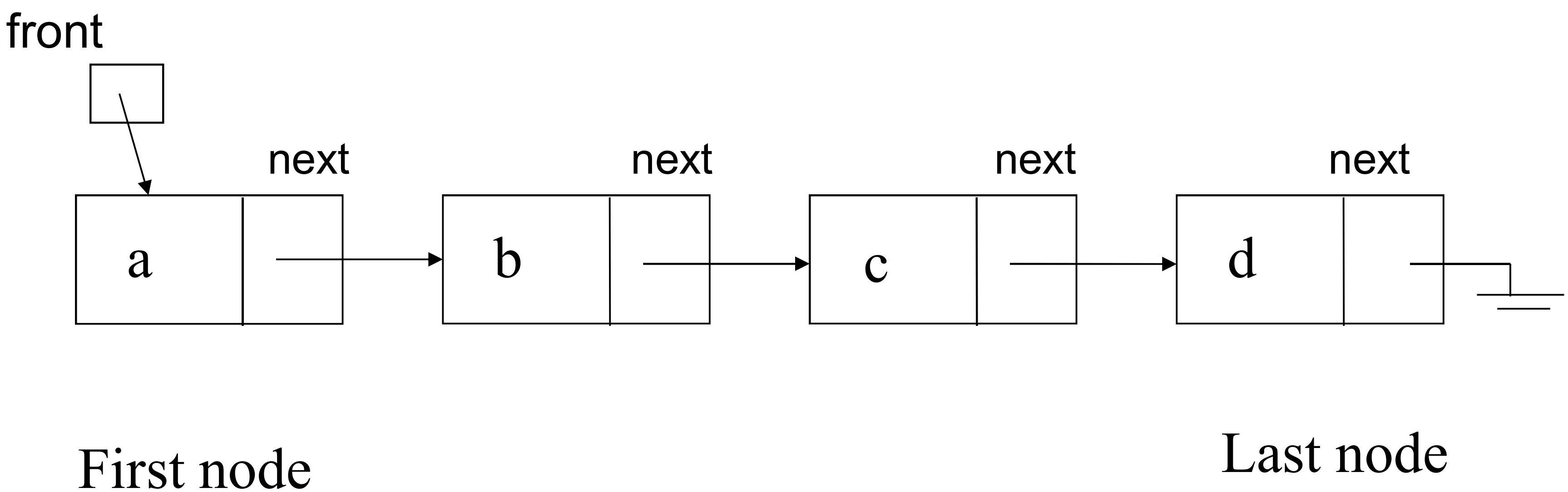
Linked List Basics

- Linked lists and arrays are similar since they both store collections of data.
- The *array's* features all follow from its strategy of allocating the memory for all its elements in one block of memory.
- *Linked lists* use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.

Linked List Basics

- Linked lists are used to store a collection of information (like arrays)
- A linked list is made of nodes that are pointing to each other
- We only know the address of the first node (head)
- Other nodes are reached by following the “next” pointers
- The last node points to NULL

Linked Lists

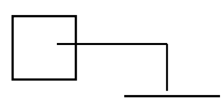


Empty List

Empty Linked list is a single pointer having the value `nullptr`.

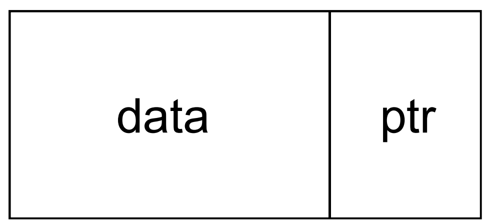
```
front = nullptr;
```

front



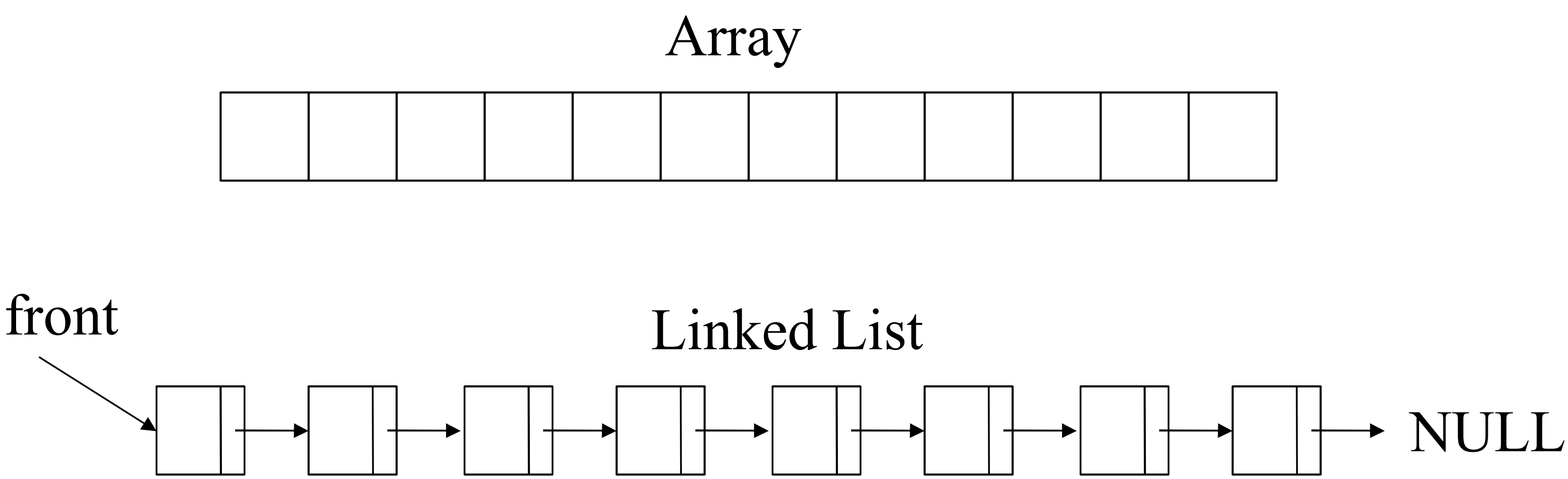
Linked List Basics

- Each node has (at least) two fields:
  - Data
  - Pointer to the next node



Linked List vs. Array

- In a linked list, nodes are not necessarily contiguous in memory (each node is allocated with a separate “new” call)
- Compare this to arrays which are contiguous



Linked List vs. Array

- **Advantages of Arrays**
  - Can directly select any element
  - No memory wasted for storing pointers
- **Disadvantages of Arrays:**
  - Fixed size (cannot grow or shrink dynamically)
  - Need to shift elements to insert an element to the middle
  - Memory wasted due to unused elements
- **Advantages of Linked Lists:**
  - Dynamic size (can grow and shrink as needed)
  - No need to shift elements to insert into the middle
  - Size can exactly match the number of elements (no wasted memory)
- **Disadvantages of Linked Lists**
  - Cannot directly select any element (need to follow ptrs)
  - Extra memory usage for storing pointers

Linked List vs. Array

- In general, we use linked lists if:
  - The number of elements that will be stored cannot be predicted at compile time
  - Elements may be inserted in the middle or deleted from the middle
  - We are less likely to make random access into the data structure (because random access is expensive for linked lists)

Linked List Applications in CS

- Implementation of stacks, queues, graphs, hash table etc.
- Dynamic memory allocation : linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- Representing sparse matrices

Applications of linked list in real world

- *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
- *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.
- *Multiplayer games* - use a circular list to swap between players in a loop.