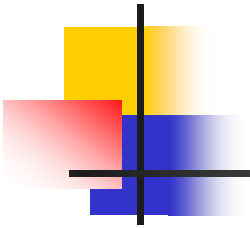
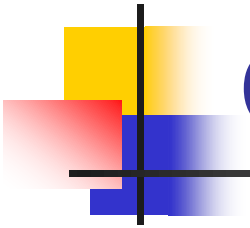
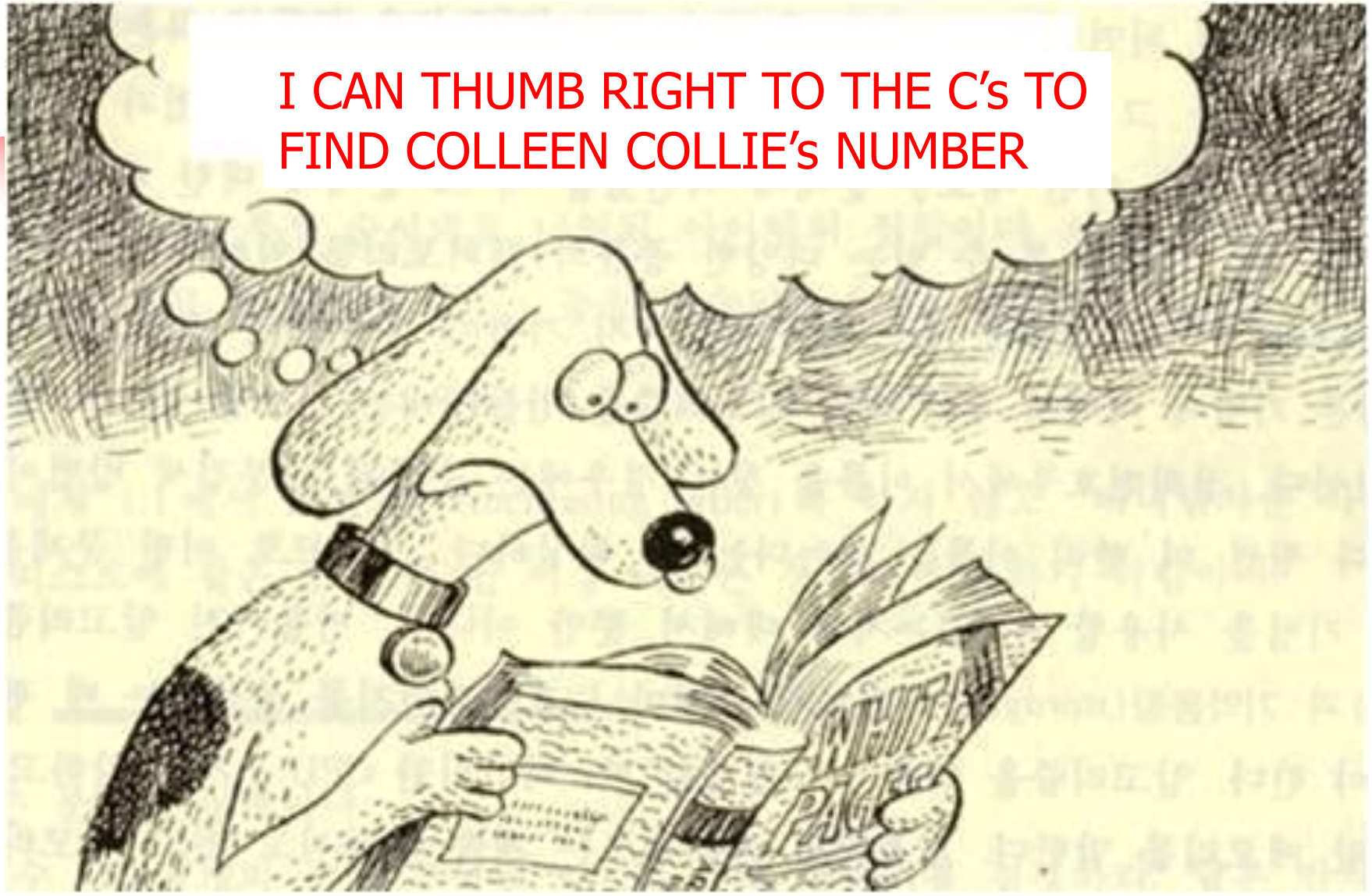


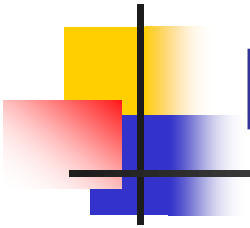
[Chapter 1]
Algorithms : Efficiency, Analysis, And Order



Goals of the Course

- Design
  - Problem solving method
  - 5 design methods: divide-and-conquer, dynamic programming, greedy approach, backtracking, branch-and-bound
- Analysis
  - Efficiency analysis through computational complexity
  - Lower bounds for sorting and searching problems
  - Intractability (NP-completeness)

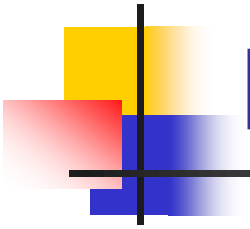

I CAN THUMB RIGHT TO THE C's TO FIND COLLEEN COLLIE's NUMBER
WHITE PAGES



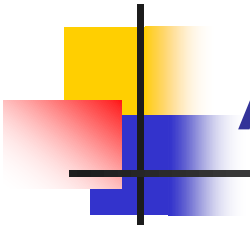
Example

- Problem
  - Determine whether the number x is in the list of S of n numbers. The answer is yes if x is in S and no if it is not.
  - Problem instance
    - S = [10, 7, 11, 5, 13, 8], n = 6, x = 8
    - Solution to this instance is, "Yes, x is in S"
- Algorithm
  - Starting with the first item in S, compare x with each item in S in sequence until x is found or S is exhausted. If x is found, answer yes; if x is not found, answer no.
- Analysis
  - Any *better* algorithm to get the same solution?

Problem Description

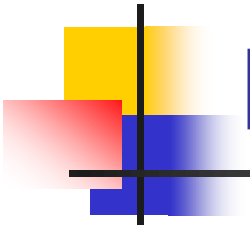


- Problem
  - May contain variables that are assigned specific values in the statement of the problem description.
- Parameters
  - Those variables are called parameters. For example, to describe a search problem, we need 3 variables: S, n, x.
- Instance
  - If those parameters are specified, we call it an instance.
  - S = [10, 7, 11, 5, 13, 8], n = 6, x = 8
- Solution to an instance of a problem is the answer to the question asked by the problem in that instance.
  - Solution to the above instance is, "yes, x is in S"



Algorithm Description

- Natural languages
- Programming languages
- Pseudo-code
  - similar, but not identical, to C++/Java.
  - Notable exceptions: unlimited array index, variable-sized array, mathematical expressions, use of arbitrary types, convenient control structure, and etc.
- In this lecture, algorithms will be represented by pseudo-code similar to C++.

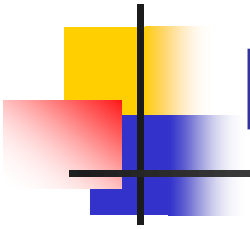


Pseudo-code vs. C++ (1/2)

- Use of arrays
  - In C++, starting at 0
  - In pseudo-code, arrays indexed by other integers are ok.
- Variable-sized array size
  - (e.g) void example (int n)

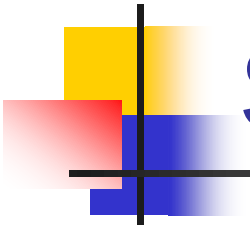
```
{  keytype S[2..n];
   ….
}
```

  - keytype S[low..high]



Pseudo-code vs. C++ (2/2)

- Mathematical expression
  - `low <= x && x <= high` ⇒ `low ≤ x ≤ high`
  - `temp = x; x = y; y = temp` ⇒ `exchange x and y`
- Use of arbitrary type
  - Index
  - Number
  - Bool
- Control structure
  - (e.g.) `repeat (n times) { … }`

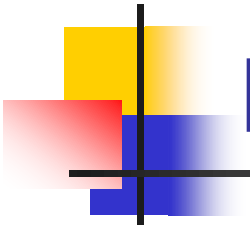


Sequential Search

- Problem
  - Is the key x in the array S of n keys?
- Inputs (parameters)
  - Positive integer n, array of keys S indexed from 1 to n.
- Outputs
  - The location of x in S. (0 if x is not in S.)
- Algorithm
  - Starting with the first item in S, compare x with each item in S in sequence until x is found or S is exhausted. If x is found, answer yes; if x is not found, answer no.

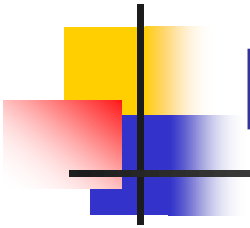
Sequential Search (Psedo-code)

```
void seqsearch(int n,                  // Input(1)
               const keytype S[],      // (2)
               keytype x,              // (3)
               index& location) {      // Output
  location = 1;
  while (location <= n && S[location] != x)
    location++;
  if (location > n)
    location = 0;
}
```



Review: sequential search

- How many comparisons for searching x in S?
  - Depends on the location of x in S
  - In worst case, we should compare ( ) times.
  - In best case,
- Any *better* algorithm to get the same solution?
  - No, we can't, unless there is an extra information in S.



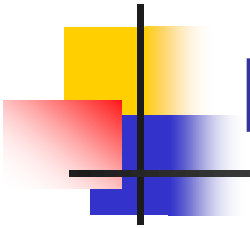
Binary Search

- Problem
  - Is the key x in the array S of n keys?
  - Determine whether x is in the sorted array S of n keys.
- Inputs (parameters)
  - Positive integer n, sorted (non-decreasing order) array of keys S indexed from 1 to n, a key x.
- Outputs
  - The location of x in S. (0 if x is not in S.)

Binary Search (Psedo-code)

```
void binsearch(int n,                  // Input(1)
               const keytype S[],      // (2)
               keytype x,              // (3)
               index& location) {      // Output
  index low, high, mid;
  low = 1; high = n;
  location = 0;

while (low <= high && location == 0) {
    mid = (low + high) / 2;    // integer div
    if (x == S[mid]) location = mid;
    else if (x < S[mid]) high = mid - 1;
    else low = mid + 1;
  }
}
```



Review: binary search algorithm

- How many comparisons for searching *x* in *S*?
  - *S* is already sorted. We know it.
  - For each statement in a while-loop, the number of searching targets will be half.
  - In worst case, we should compare ( ) times.
    - (e.g.) n = 32. S[16], S[16+8], S[24+4], S[28+2], S[30+1]. S[32]
  - In best case, trivial to answer.
- Any *better* algorithm to get the same solution?
  - Hmmm. Very goooooood question.
  - But, wait to see the answer until we get to Chap. 7-8.

Table 1.1 The number of comparisons done by Sequential Search and Binary Search when x is larger than all the array items

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

The Fibonacci Sequence

- Problem
  - Determine the *n*-th term in the Fibonacci sequence.

$$f_0 = 0$$

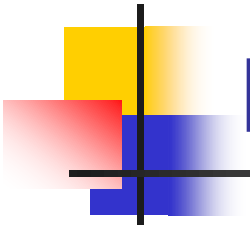
$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2$$

- Inputs
  - A non-negative integer n.
- Outputs
  - The *n*-th term of the Fibonacci sequence.

1st Solution to the n-th Fib. Number (Recursive version)

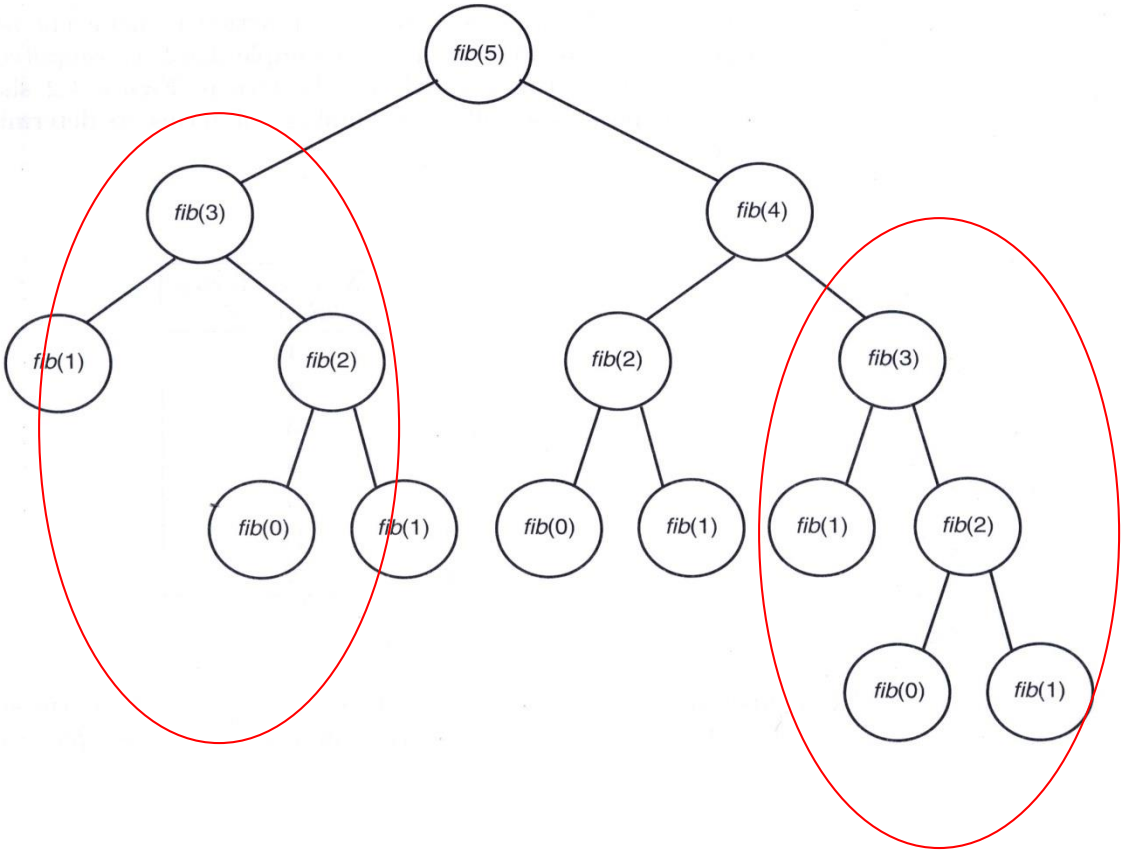
```
int fib (int n) {
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```



Review: recursive solution to Fib.

- How many computations to get the n-*th* number?
  - $2^{n/2}$.
  - No difference between the worst and best cases.
  - Awkwardly slow algorithm. Why?
- Any *better* algorithm to get the same solution?
  - If we know the problem exactly, we can solve it.
  - What is the cause of the slowness?

The recursion tree when computing the fifth Fibonacci term.



Counting the number of calls needed in fib(n)

$T(n)$ = the number of terms in the recursion tree for n

$T(0) = 1$

$T(1) = 1$

$T(n) = T(n - 1) + T(n - 2) + 1$ for $n \geq 2$

$> 2 \times T(n - 2)$ because $T(n - 1) > T(n - 2)$

$> 2^2 \times T(n - 4)$

$> 2^3 \times T(n - 6)$

$\cdots$

$> 2^{n/2} \times T(0)$

$= 2^{n/2}$

Theorem 1.1 $T(n) > 2^{n/2}$

Proof by mathematical induction

Induction base:

$$T(2) = T(1) + T(0) + 1 = 3 > 2 = 2^{2/2}$$

$$T(3) = T(2) + T(1) + 1 = 5 > 2.83 \approx 2^{3/2}$$

Induction Hypothesis:

For all integer m, suppose $T(m) > 2^{m/2}$ holds.

Induction Step: To show that $T(n) > 2^{n/2}$….

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

$$> 2^{(n - 1)/2} + 2^{(n - 2)/2} + 1, \text{ according to I.H.}$$

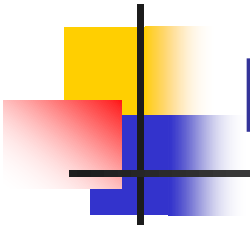
$$> 2^{(n - 2)/2} + 2^{(n - 2)/2}$$

$$= 2 \times 2^{(n / 2)-1}$$

$$= 2^{n / 2}$$

2nd Solution to the n-th Fib. Number (Iterative version)

```
int fib2 (int n) {
   index i;
   int f[0..n];
   f[0] = 0;
   if (n > 0) {
      f[1] = 1;
      for (i = 2; i <= n; i++)
         f[i] = f[i-1] + f[i-2];
   }
   return f[n];
}
```

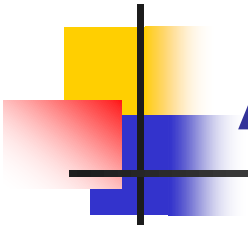


Review: iterative solution to Fib.

- Iterative version is much faster than recursive one
  - Why faster?
  - How faster
- Time complexity
  - $T(n) = n + 1$

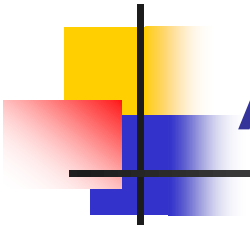
A comparison of two Fib. Algorithms

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 $\mu s^{\dagger}$
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years



Analysis of Algorithms (1/2)

- Time complexity analysis
  - Determination of how many times the basic operation is done for each value of the input size.
  - Should be independent of CPU, OS, Programming languages…
- Metrics
  - Basic operation
    - Comparisons, assignments, etc.
  - Input size
    - The number of elements in an array
    - The length of a list
    - The number of columns and rows in a matrix
    - The number of vertices and edges in a graph



Analysis of Algorithms (2/2)

- Every-case analysis
- Worst-case analysis
- Average-case analysis
- Best-case analysis

Every-case analysis

- Regardless of the values of the numbers in the array, if there are same number of computational passes in a loop, then the basic operation is always done n times, $T(n) = n$.
  - (e.g.) Addition algorithm for n integers in an array

```
number sum (int n, const number S[]) {
  index i;
  number result;

  result = 0;
  for (i = 1; i <= n; i++)
    result = result + S[i];
  return result;
}
```

Every-case Analysis (Cont'd)

- (e.g.) Exchange sort
  - Basic operation: the comparison of *S[i]* and *S[j]*
  - Input size: the number of items to be sorted

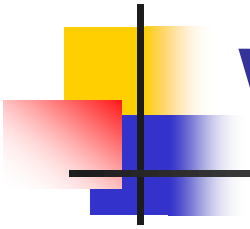
```
void exchangesort (int n, keytype S[]) {
  index i, j;

  for (i = 1; i <= n-1; i++)
    for (j = i+1; j <= n; j++)
      if (S[j] < S[i])
        exchange S[i] and S[j];
}
```

Every-case Analysis (Cont'd)

- Time complexity analysis for *Exchangesort()*
  - For each j-th loop, the if-statement will be executed once.
  - The total number of if-statement
    - $i = 1$: $n-1$ times
    - $i = 2$: $n-2$ times
    - $i = 3$: $n-3$ times
    - $i = n-1$: $n-(n-1)$ times
    - Therefore,

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



Worst-case analysis

- Sequential search
  - Basic operation: (`S[location] != x`)
  - Input size: the number of items in an array, *n*
  - Worst case happens if *x* is in the last element of *S[n]*
  - Therefore, *W(n) = n*.
  - Every-case analysis is not possible for sequential search.

Average-case analysis

- Sequential search
  - Basic operation: (`S[location] != x`)
  - Input size: the number of items in an array, `n`
  - Average case
    - Assume each item in the array is distinctive
    - Case 1: when x is always in *S[n]*
      - The probability that *x* is the *k*-th element in *S[n]* $= 1/n$
      - The number of operations if *x* is the *k*-th = *k* times
      - Therefore,

$$A(n) = \sum_{k=1}^{n} k \times \frac{1}{n} = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Average-case analysis (Cont'd)

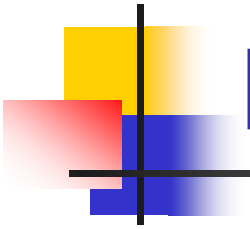
- Case 2: *x* is either in *S[n]* or not in *S[n]*
  - The probability that *x* is in *S[n]*: *p*
    - The probability that *x* is in the *k*-th position of *S[n]* = *p/n*
    - The probability that x is not in *S[n]* = *1 – p*
  - Therefore , $A(n) = \sum_{k=1}^{n} (k \times \frac{p}{n}) + n(1-p)$

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

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

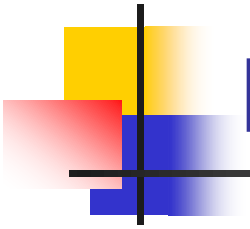
  - $p = 1 \Rightarrow A(n) = (n+1)/2$

    $p = 1/2 \Rightarrow A(n) = 3n/4 + 1/4$



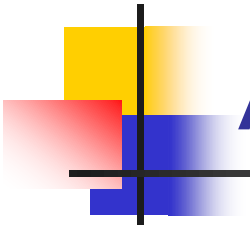
Best-case analysis

- Sequential search
  - Basic operation: (`S[location] != x`)
  - Input size: the number of items in an array, `n`
  - Best case happens if *x* is the first element of *S[n]*, i.e. *S[1]*
  - Therefore, *B(n) = 1*.



Review: time complexity analysis

- Among four possible analysis, ECA, WCA, ACA, and BCA, which one is the right one?
- Think about …
  - you are working for a nuclear power plant.
  - what if you are working for an internet shopping mall
- Which one do you think is the most useless?
- Which one do you think is the hardest to analyze?



Analysis of Correctness

- Efficiency analysis vs. Correctness analysis
- In this course, when I say an algorithm analysis, I mean an efficiency analysis
- We can also analyze the correctness of an algorithm by developing a mathematical proof that the algorithm actually does what it is supposed to do.
- An algorithm is *incorrect*…
  - if it does not stop for a given input or
  - if it gives a wrong answer for an input.