Search Algorithms

Efficiency and Analysis

Algorithm selection

- Algorithms are ordered lists of steps for solving a problem
- Algorithms are also abstractions of the programming process
- Often there are many different algorithms for solving the same problem

Algorithms: main considerations

- **How fast will it run**
- **How much memory will it use**
- **How easy is it to implement**
- **The choice depends on the software**
  - It may mean your algorithm is not the most efficient

Program efficiency

- **Computer scientists analyze algorithms for their efficiency**
  - In a machine independent manner
  - By focussing on the critical operations performed
  - Like the number of comparisons
    - (for sorting routines)

Rate of growth

- **The most important aspect of algorithm efficiency**
- **Two algorithms may be just as efficient for small data sets but differ greatly for large ones.**

Asymptotic Efficiency

- **The** asymptotic efficiency **of an algorithm**
  - describes its relative efficiency as *n* gets very large.
- **Often called the "order of complexity"**
- **"Big O" - O()**
- **Example: Sequential search is O(n)**

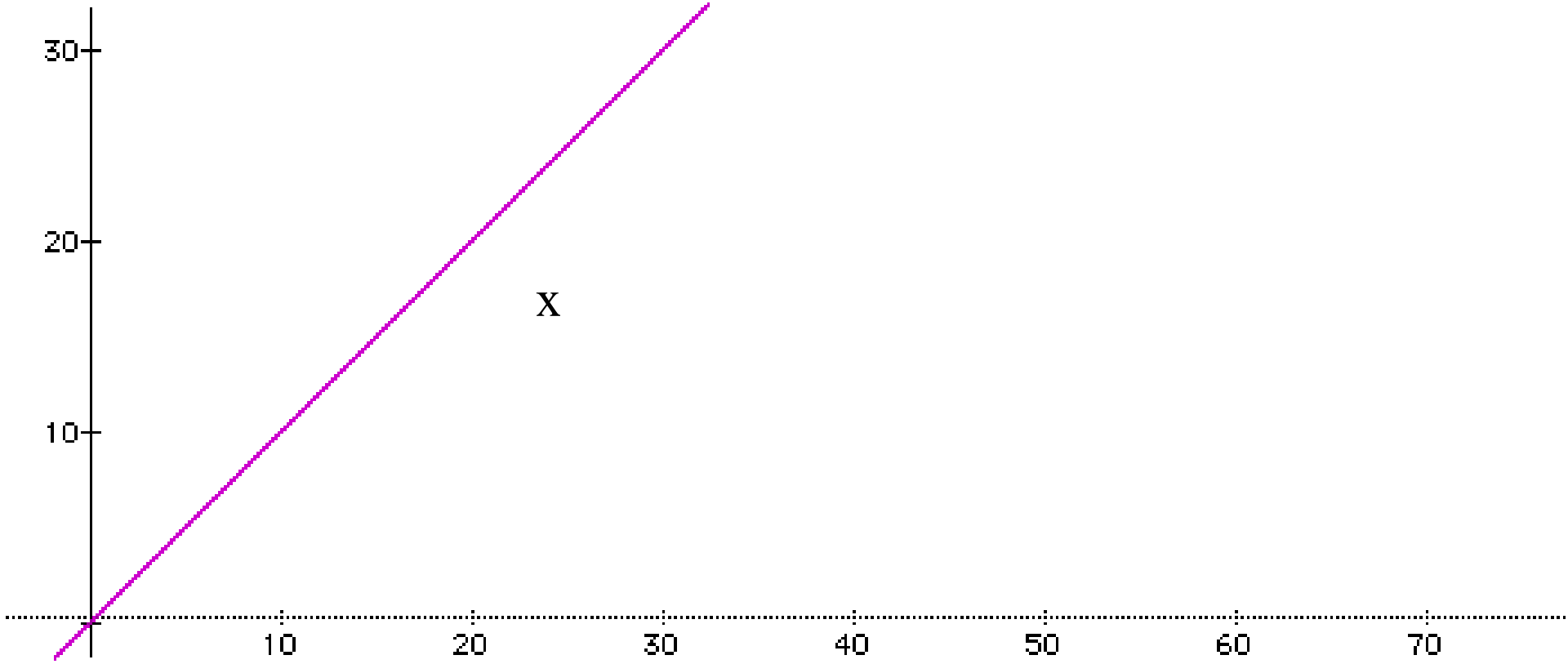
Exercise

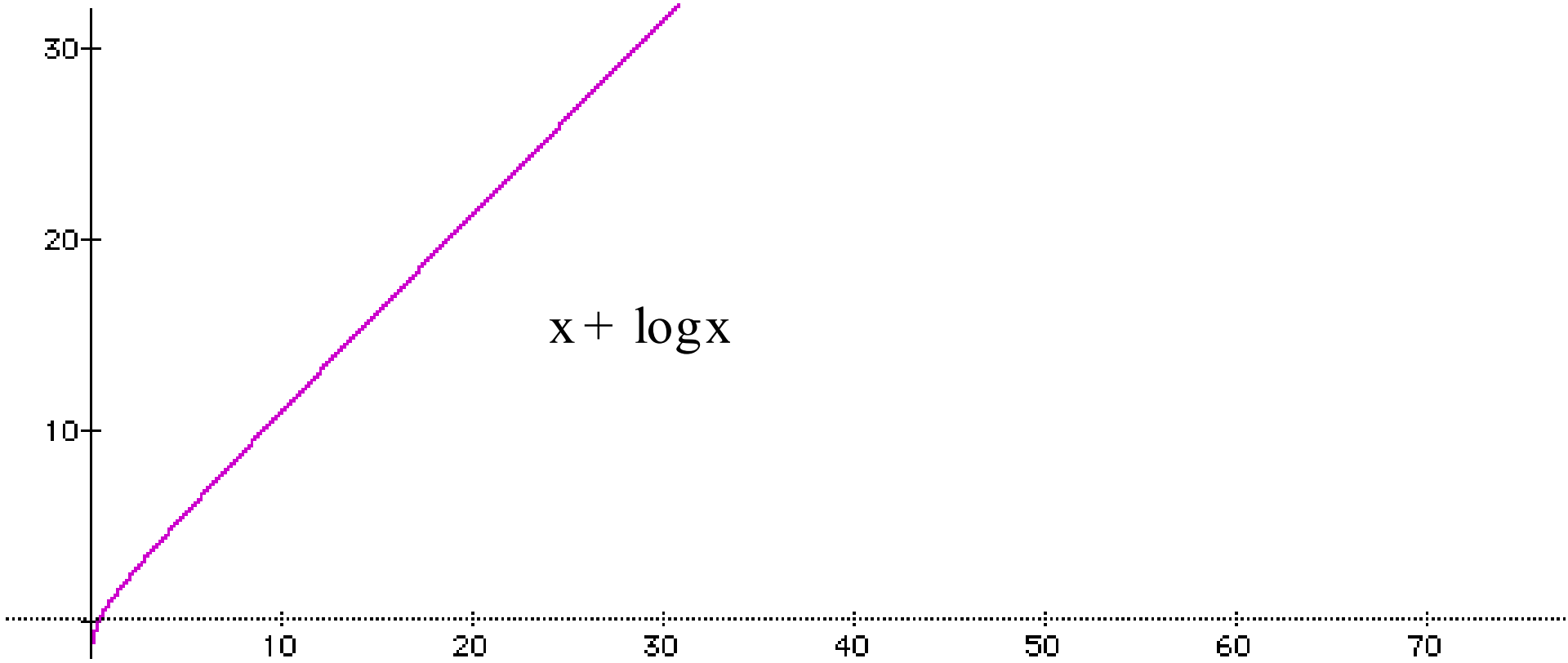
- Rank the asymptotic orders of the following functions, from highest to lowest.
  - (You may wish to graph some or all of them.)

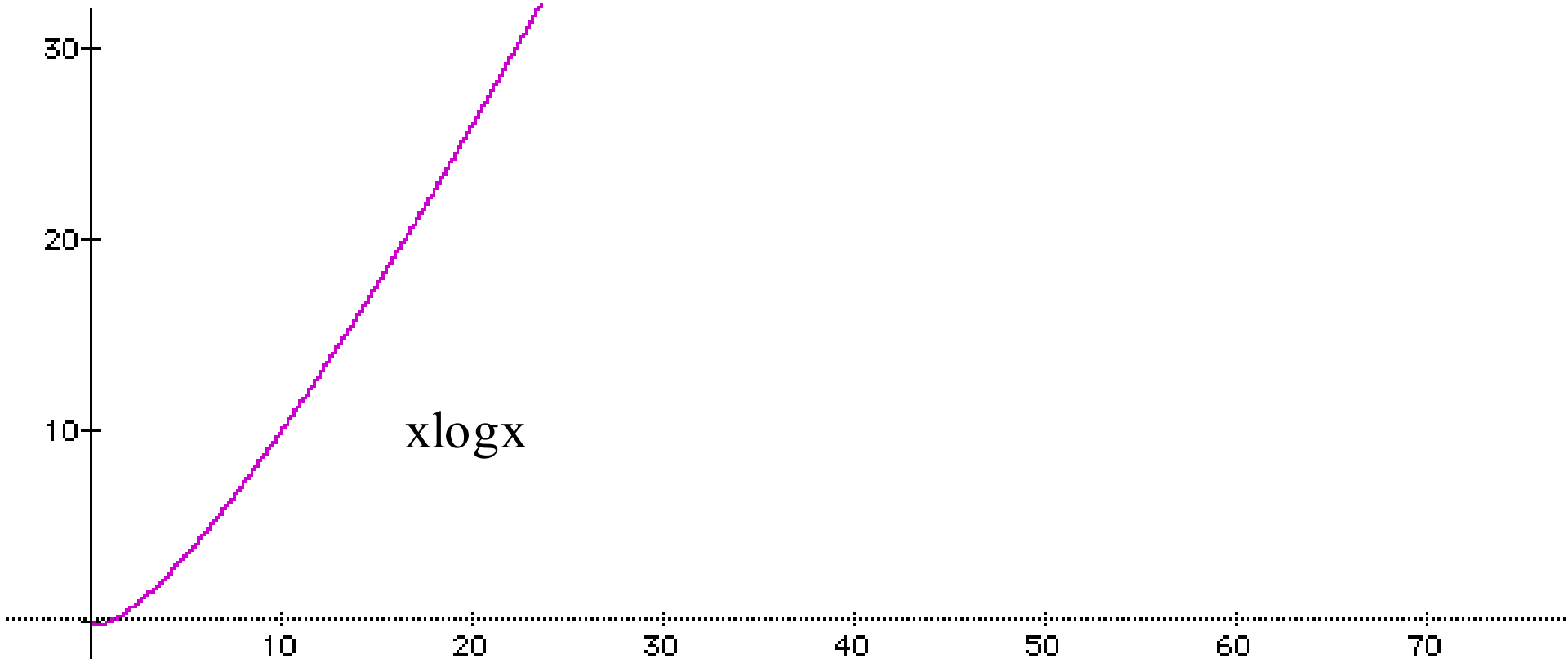
$3n^3 + 2n + 1000$ $30n + 20n + 10n$ $60n$

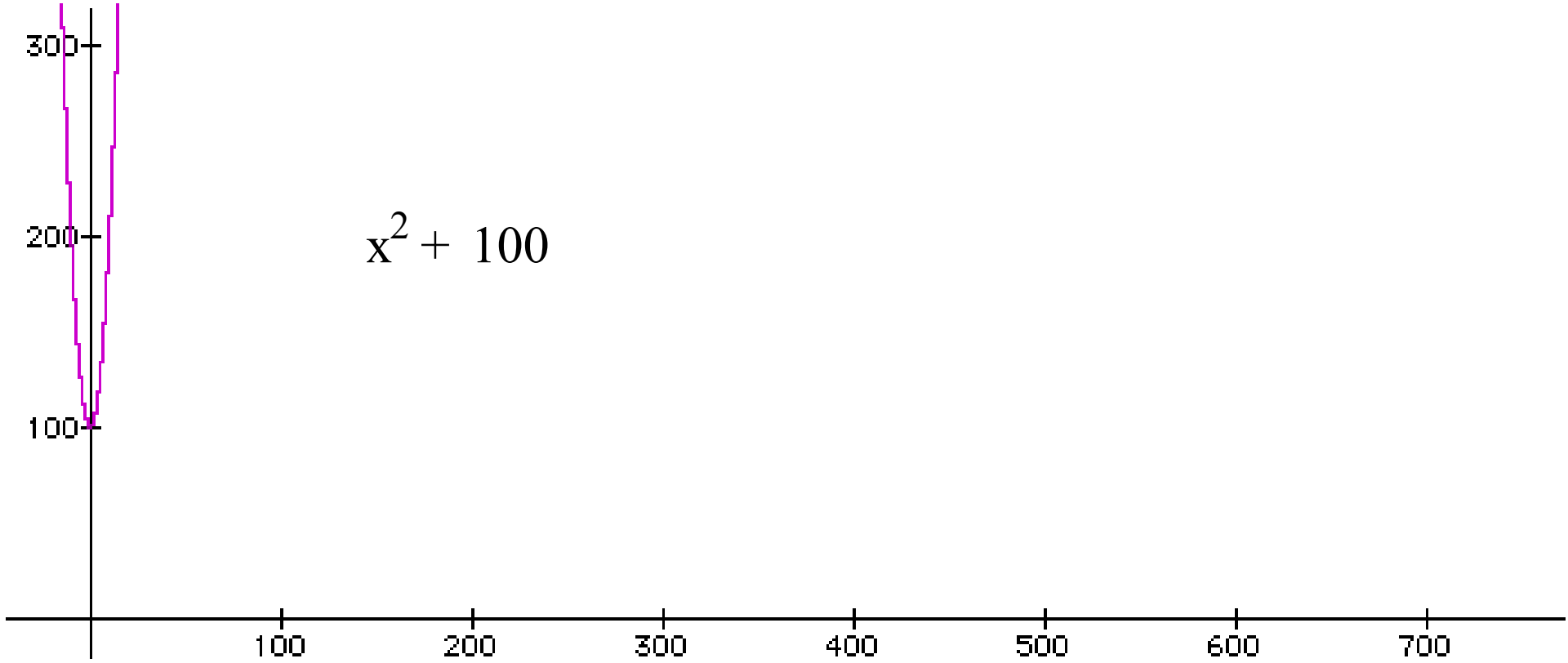
100 $n^2 + 100$ $n \log n$

$2^n + n^2$ $n + \log n$

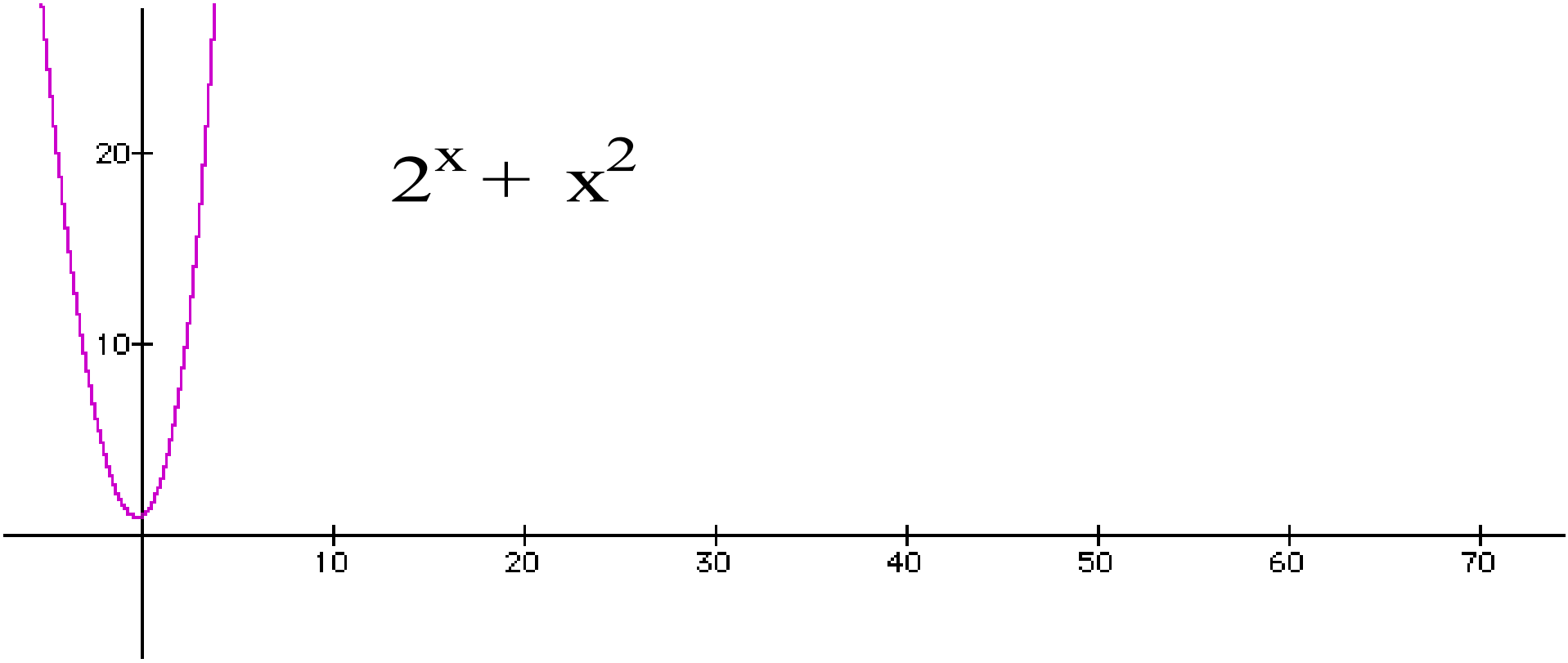

30
20
10
X
10
20
30
40
50
60
70

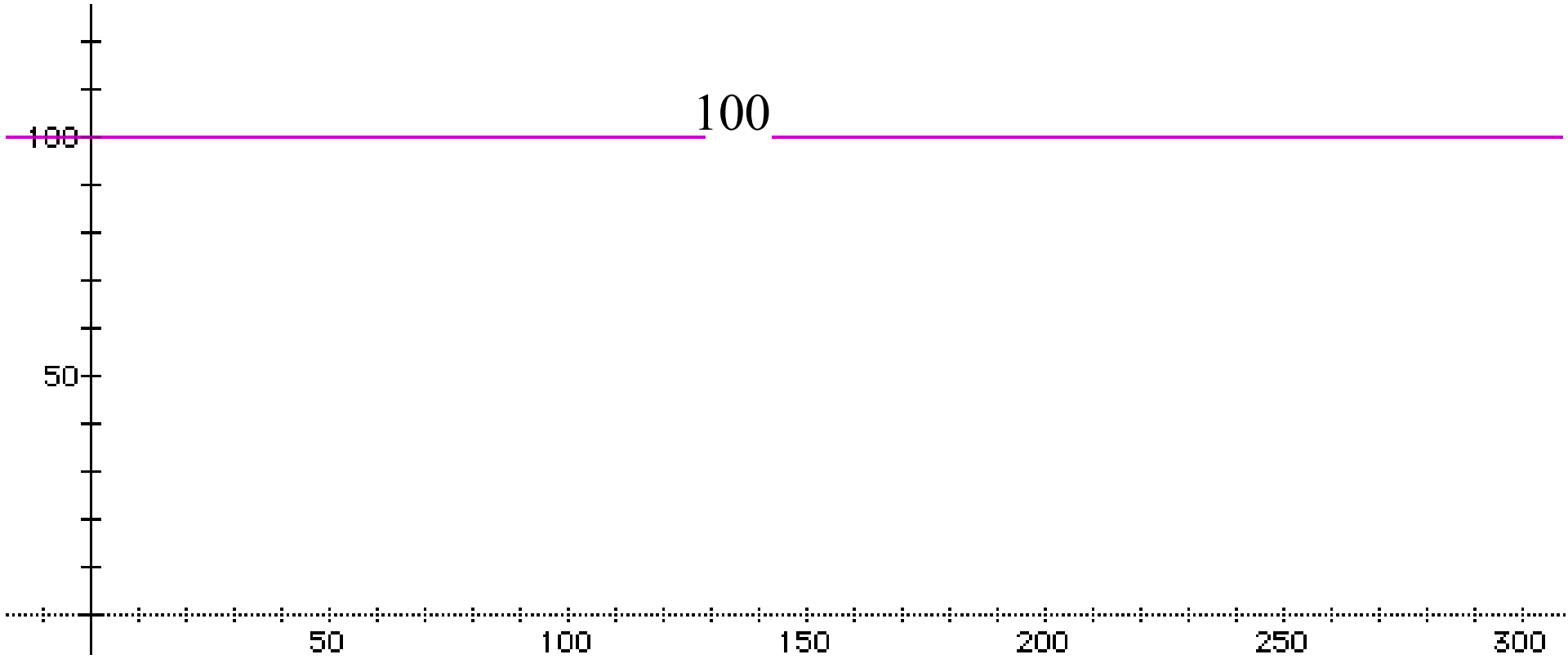

30
20
10
x + logx
10
20
30
40
50
60
70

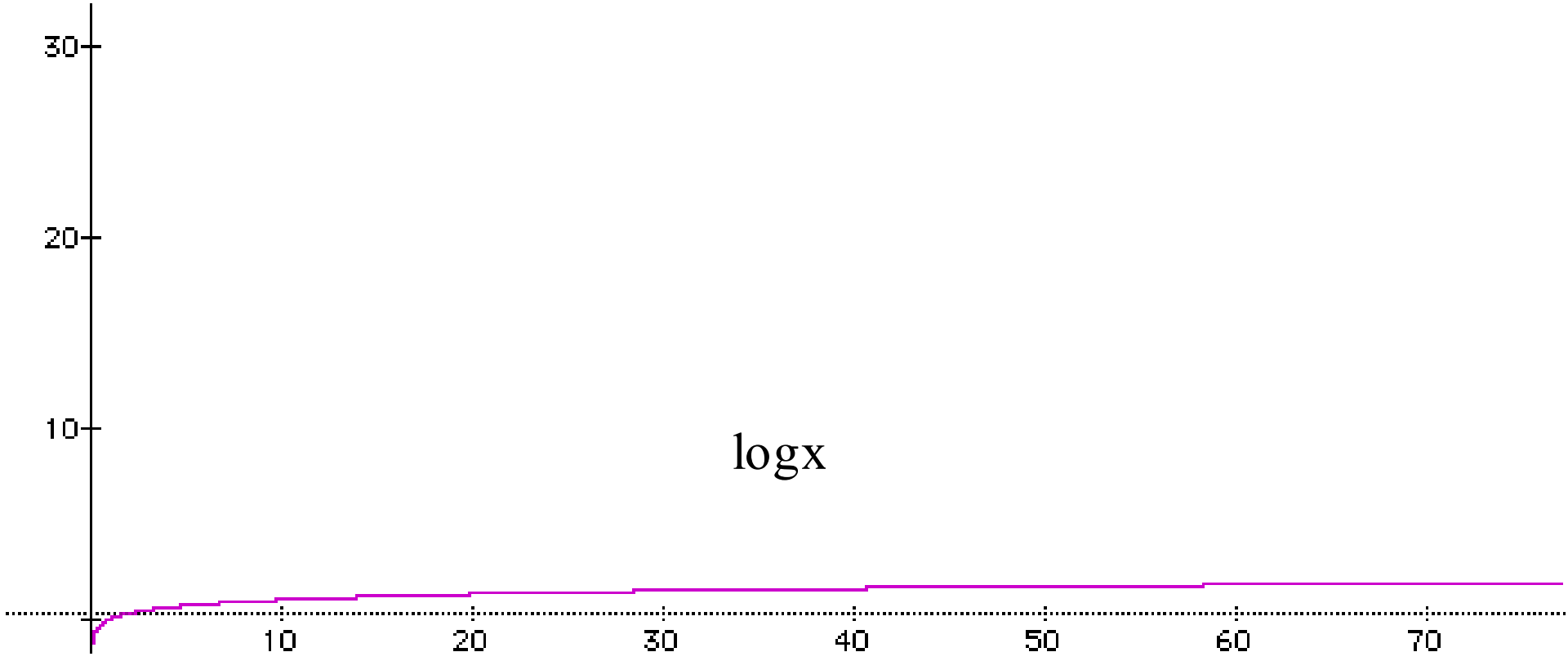

30
20
10
xlogx
10
20
30
40
50
60
70

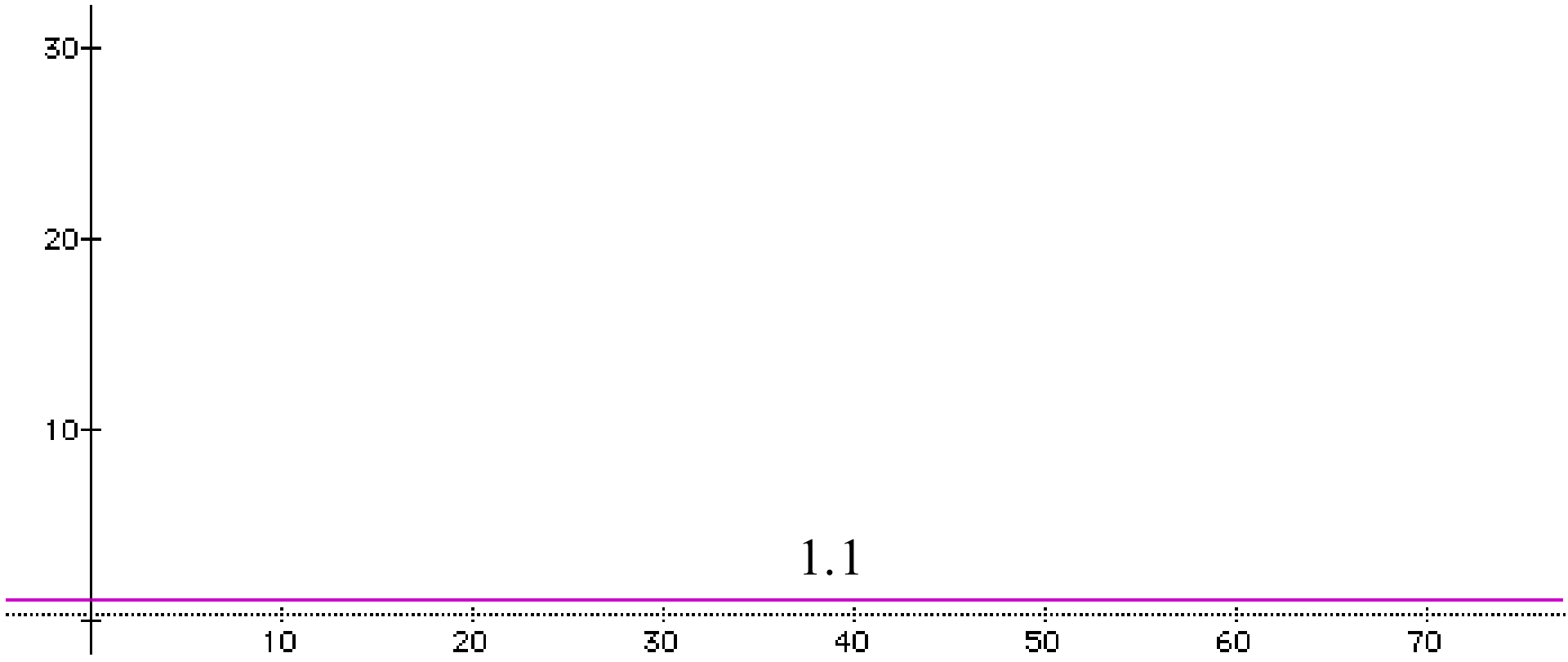


$x^2 + 100$


$2^x + x^2$
20
10
10
20
30
40
50
60
70


100
100
50
50
100
150
200
250
300


30
20
10
logx
10
20
30
40
50
60
70


30
20
10
1.1
10
20
30
40
50
60
70

Analyzing the Linear Search

- Assumes a list of n elements
- Assumes a search key
- A linear search looks for the target key value by proceeding in sequential fashion through the list

Sample record format

key	data

Linear Search algorithm

- For each item in the list
- if the item's key matches the target,
- stop and report "success"
- Report "failure"

Linear Search

```
int linearSearch(int a[], int n, int target)
{
   int i;
   for (i = 0; i < n; i++)
      if (a[i] == target) // key comparison
         return i;
   return -1; // use -1 to indicate failure
}
```

Analysis

- **Speed of an algorithm is measured by counting key comparisons**
- **Best case is 1 comparison**
- **Worst case is n comparisons**
- **Average case is n/2 comparisons**
  - if the target is in the list

Analysis (continued)

- What if the target we are looking for has only a 50% chance of being in the list.
- The complexity must account for both targets that can be found and targets that cannot.
- For targets that can be found it is n/2
- For targets that cannot be found it is n
- For targets that can be found only 50% of the time it is: 1/2*n/2 (found) + 1/2*n (not found)
- The order of complexity therefore is: n/4 + n/2 = 3n/4

Polynomial expressions

- All polynomial equations are made up of a set of terms.
- Usually with coefficients and exponents
- The fastest growing term is called the term of the “highest order”
- A linear search, like the last example, has only one term to describe it: 3n/4
- This can be written as (3/4)n
- 3/4 is the coefficient, n is the exponent
- n is also the highest order term

From polynomials to Big-O

- A linear function is one whose growth is tied to n
- In other words, n is the highest order polynomial
- The growth of the expression is tied to the highest order term and is called the “order of magnitude”
- Order of magnitude is expressed as Big-O
- For linear functions, Big-O is n
- We express this as O(n)
- To find out what the order of an expression is, look for the highest order term.

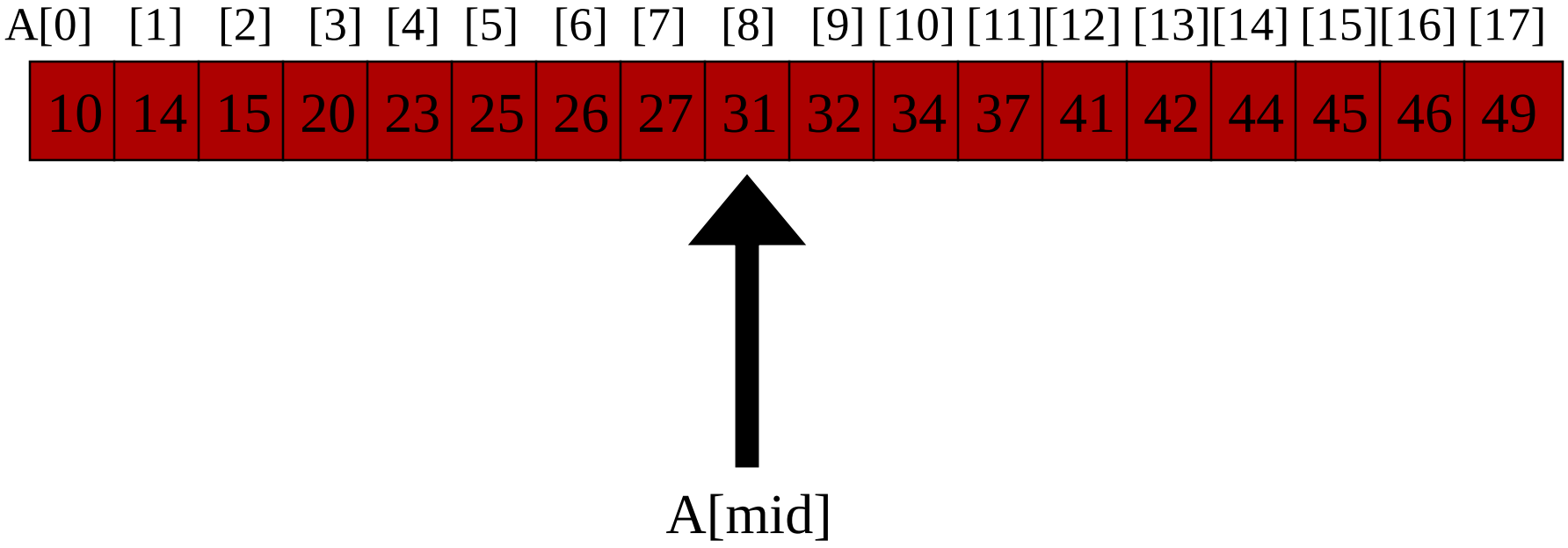
Graphing Big-O

- The x axis corresponds to the number of elements in the list
- The y axis is the number of critical operations required by an algorithm
- The order of complexity is the highest order term in the analysis because it is the one that increases the fastest
- Constant multipliers are ignored
- Example: n+100 is graphed as O(n)
- For a linear search n/2 is also O(n)
- Similarly, from last example, 3n/4 is O(n)

Binary Search

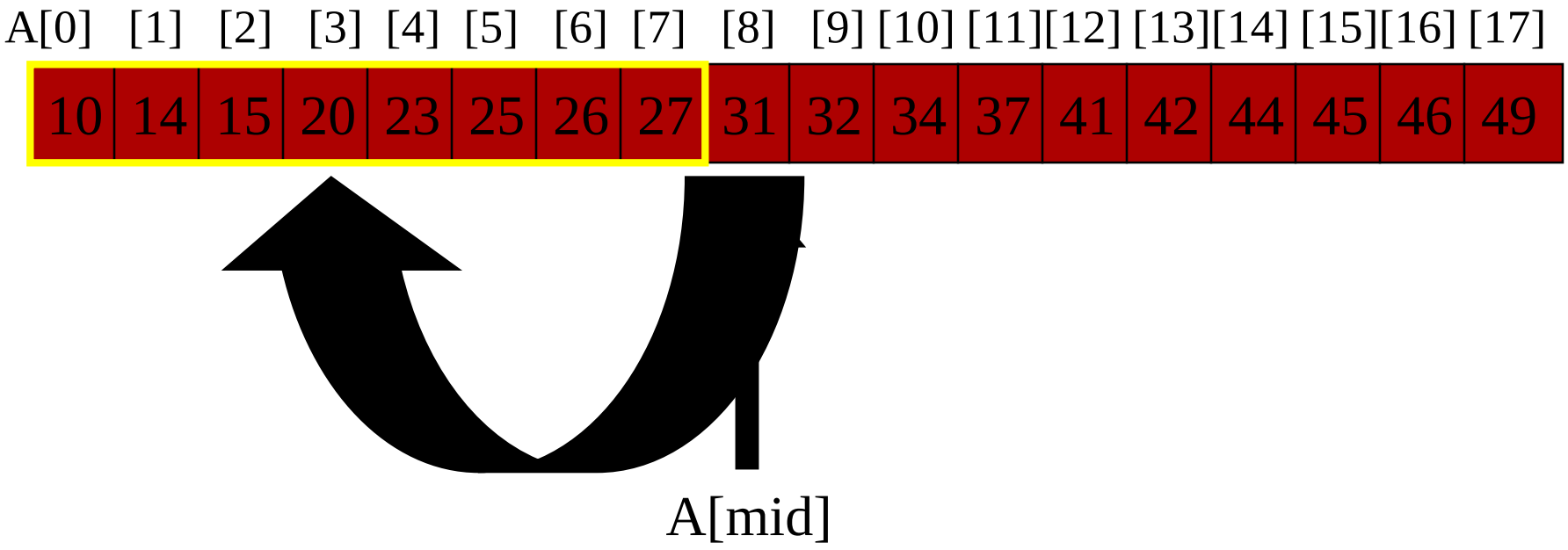
- **Repeatedly divides the list in half and in half again**
- **The result is a great improvement on linear searching**
- **What is the asymptotic efficiency?**
  - Best case is 1 comparison
  - Worst case is $\log_2 n$
- **Logarithmic complexity is much faster than linear**
  - Even if we use the worst case

Binary search strategy



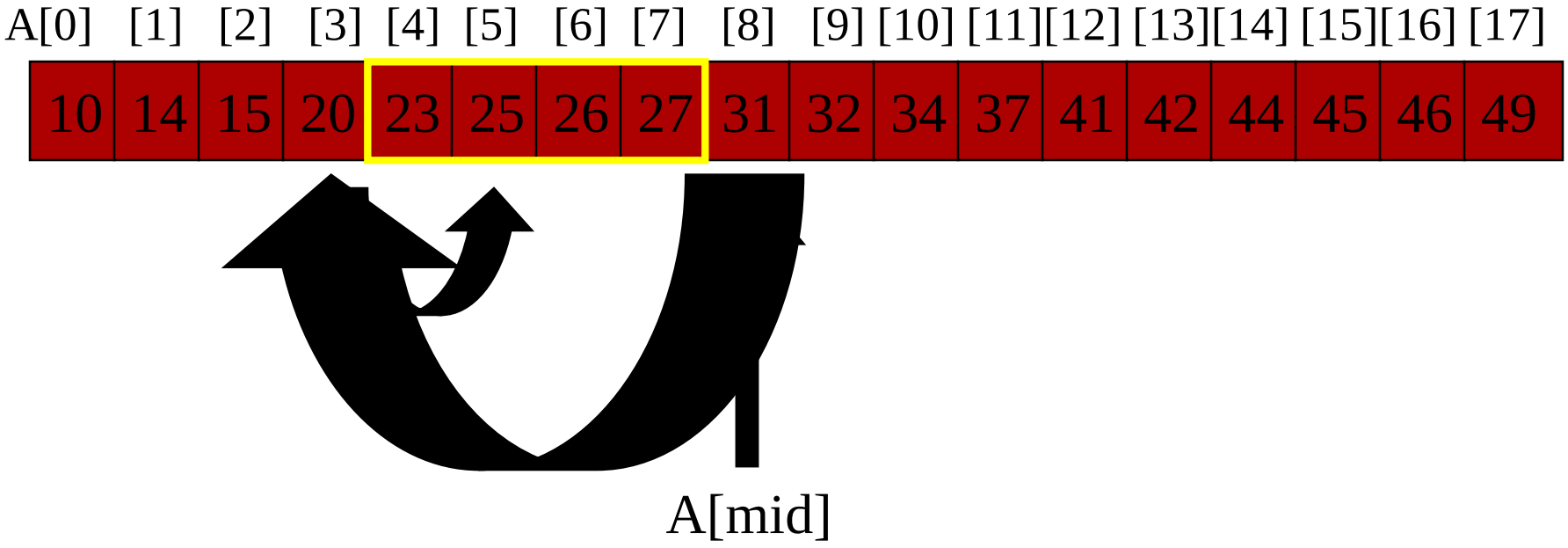
Searching for 26

Binary search strategy



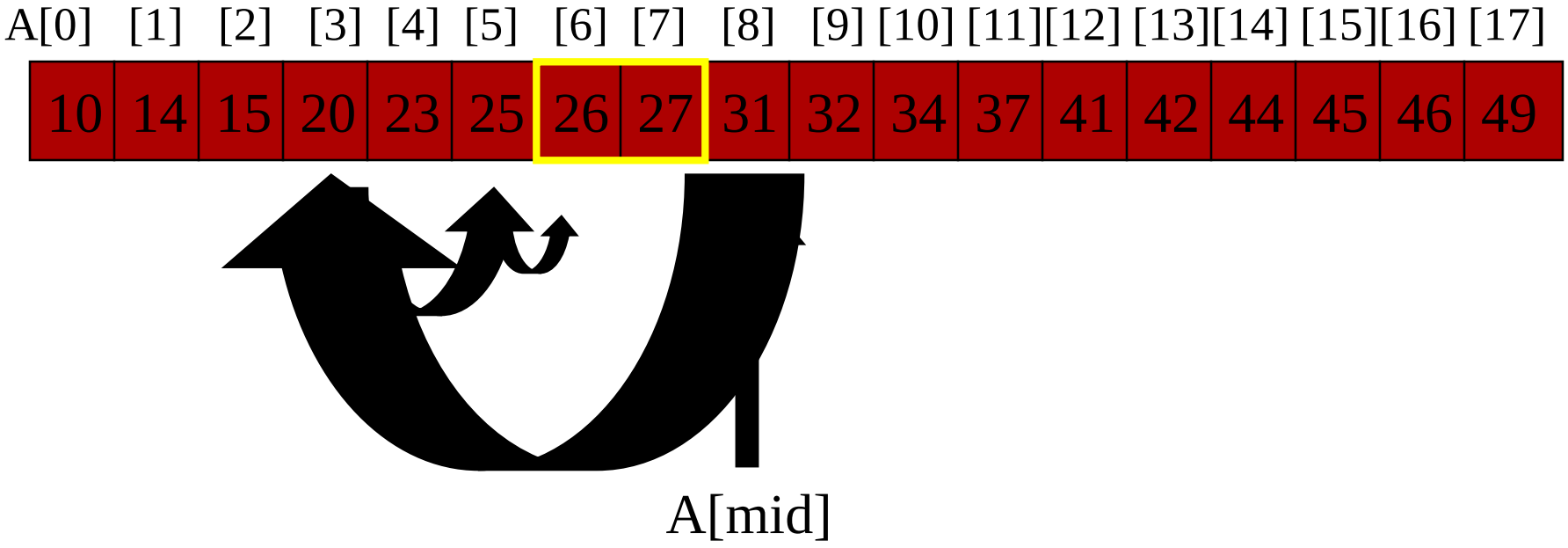
Searching for 26

Binary search strategy



Searching for 26

Binary search strategy

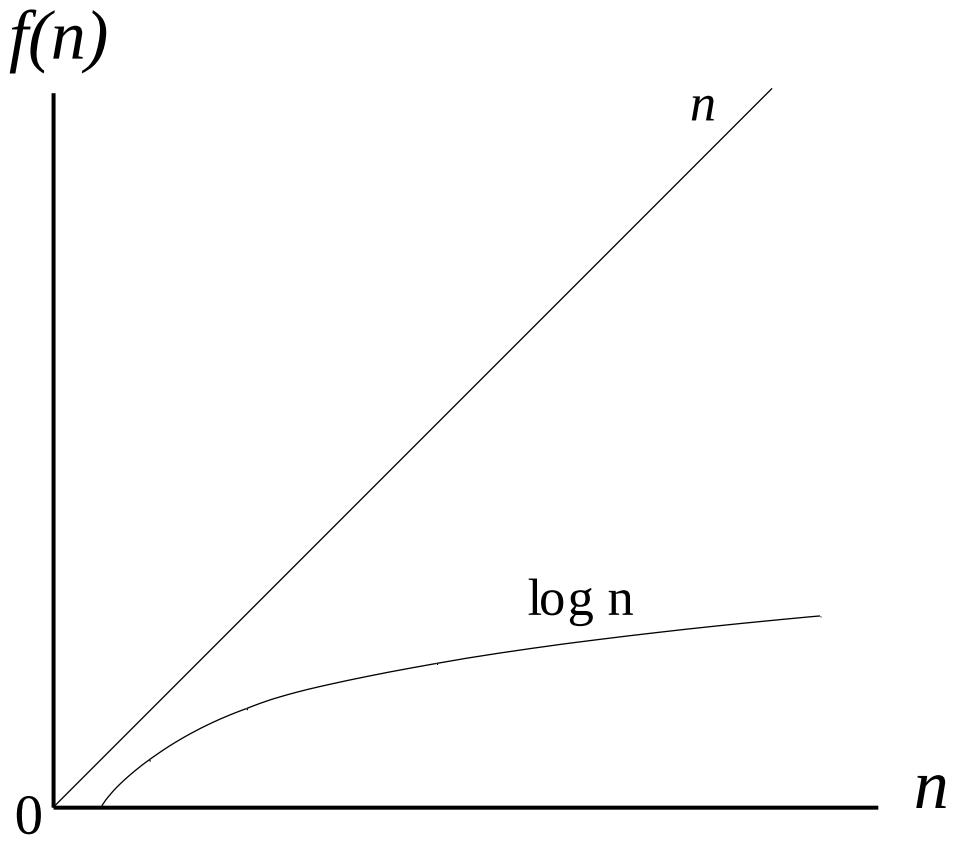


Searching for 26

The growth of the base-2 log function

n	n (as power of 2)	$\log_2 n$
16	2^4	4
256	2^8	8
4,096	2^{12}	12
65,536	2^{16}	16
1,048,576	2^{20}	20

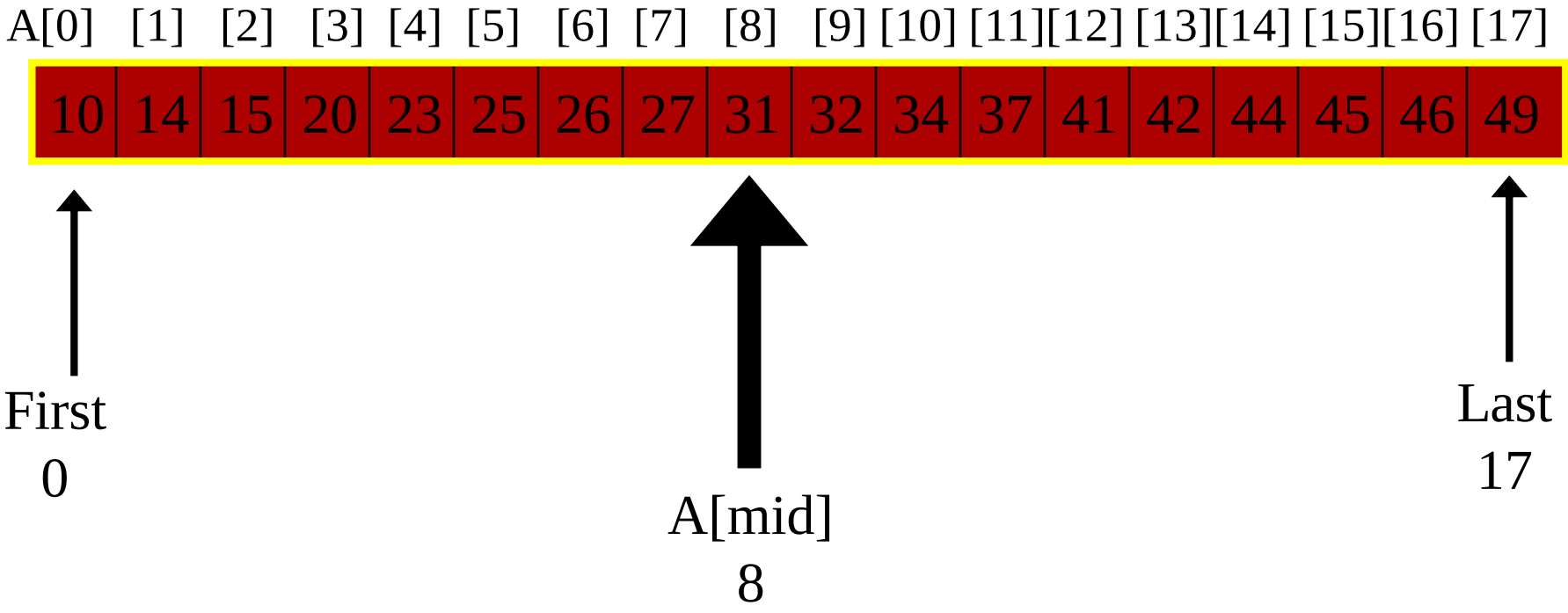
Graph illustrating relative growth of linear and logarithmic functions



Defective Binary Search

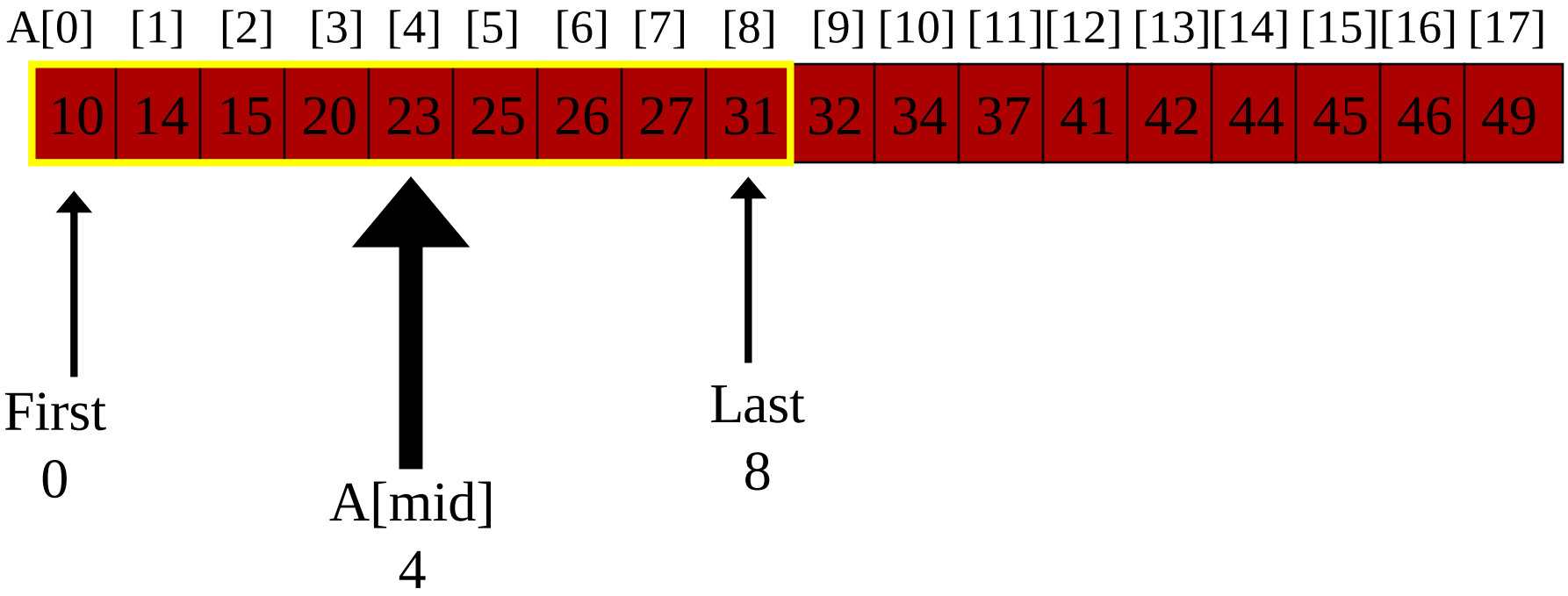
```
int binarySearch(int a[], int n, int target)
{ // Precondition: array a is sorted in ascending order from a[0] to a[n-1]
   int first(0), last(n - 1), int mid;
   while (first <= last) {
      mid = (first + last)/2;
      if (target == a[mid])
         return mid;
     else if (target < a[mid])
         last = mid;
      else // must be that target > a[mid]
         first = mid; }
   return -1; // use -1 to indicate item not found
}
```

Binary search strategy



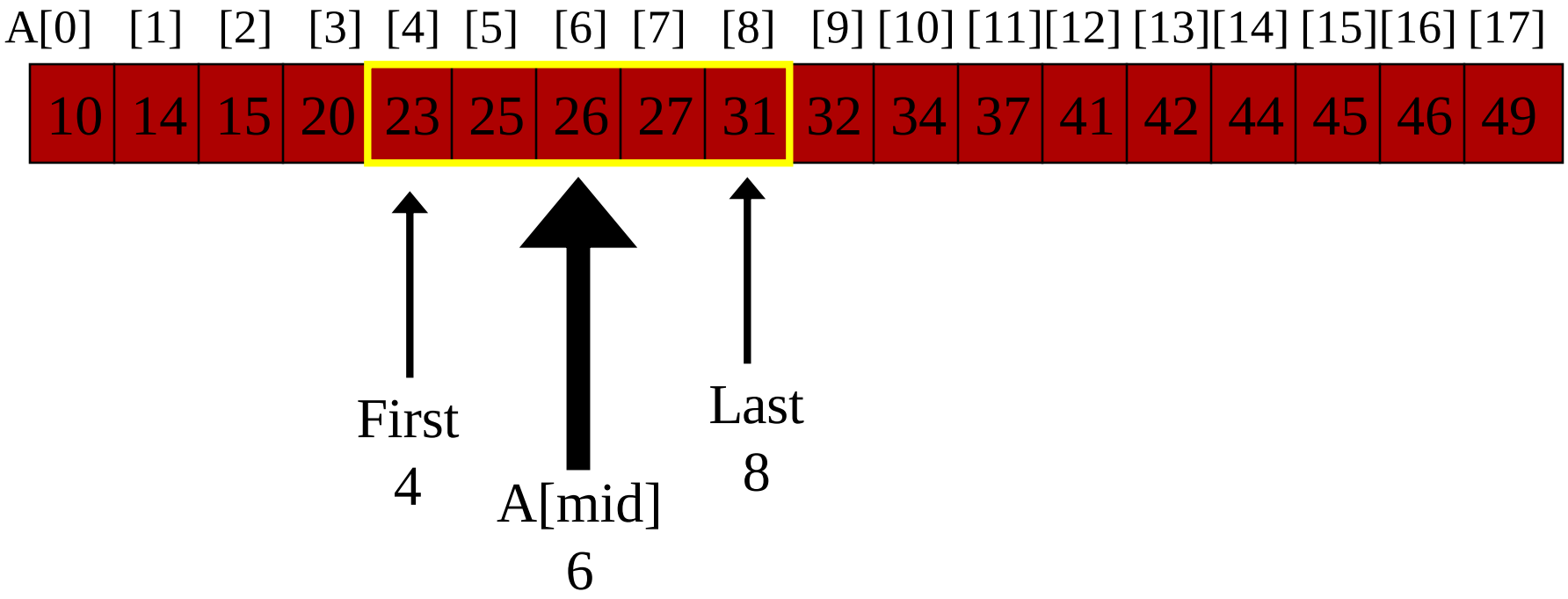
Searching for 26

Binary search strategy



Searching for 26

Binary search strategy

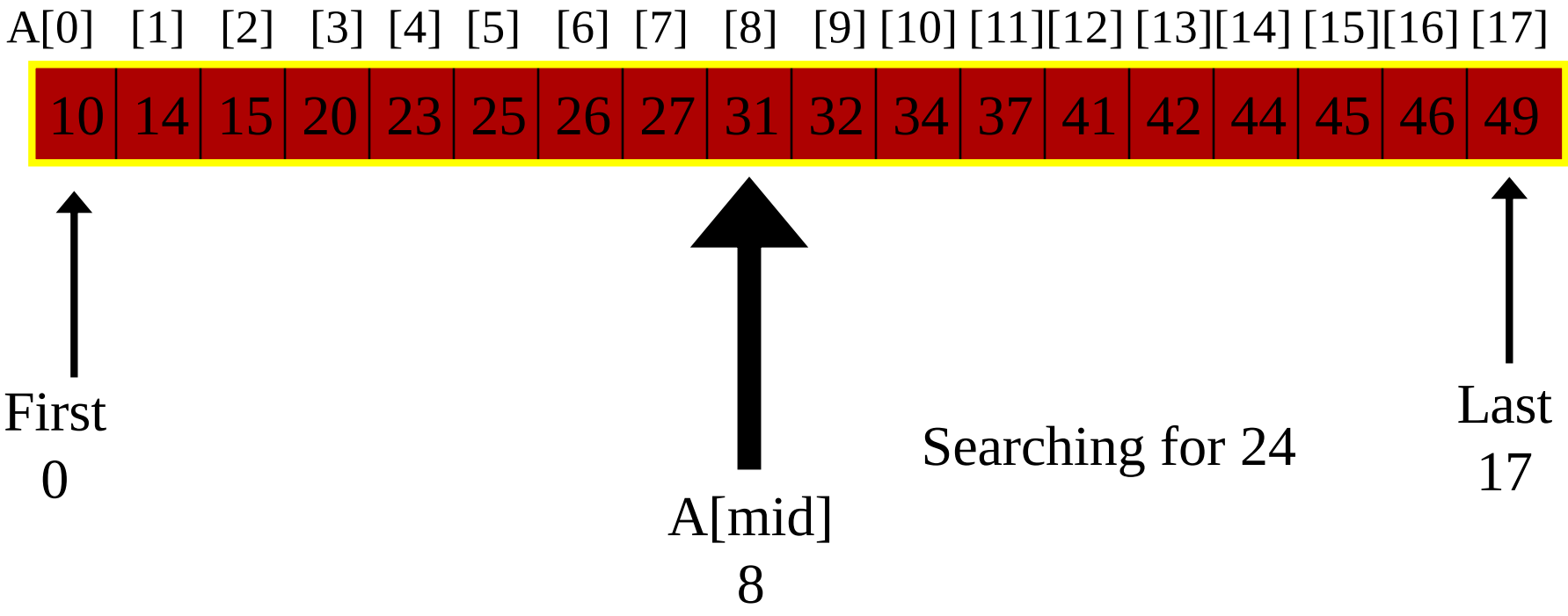


Searching for 26

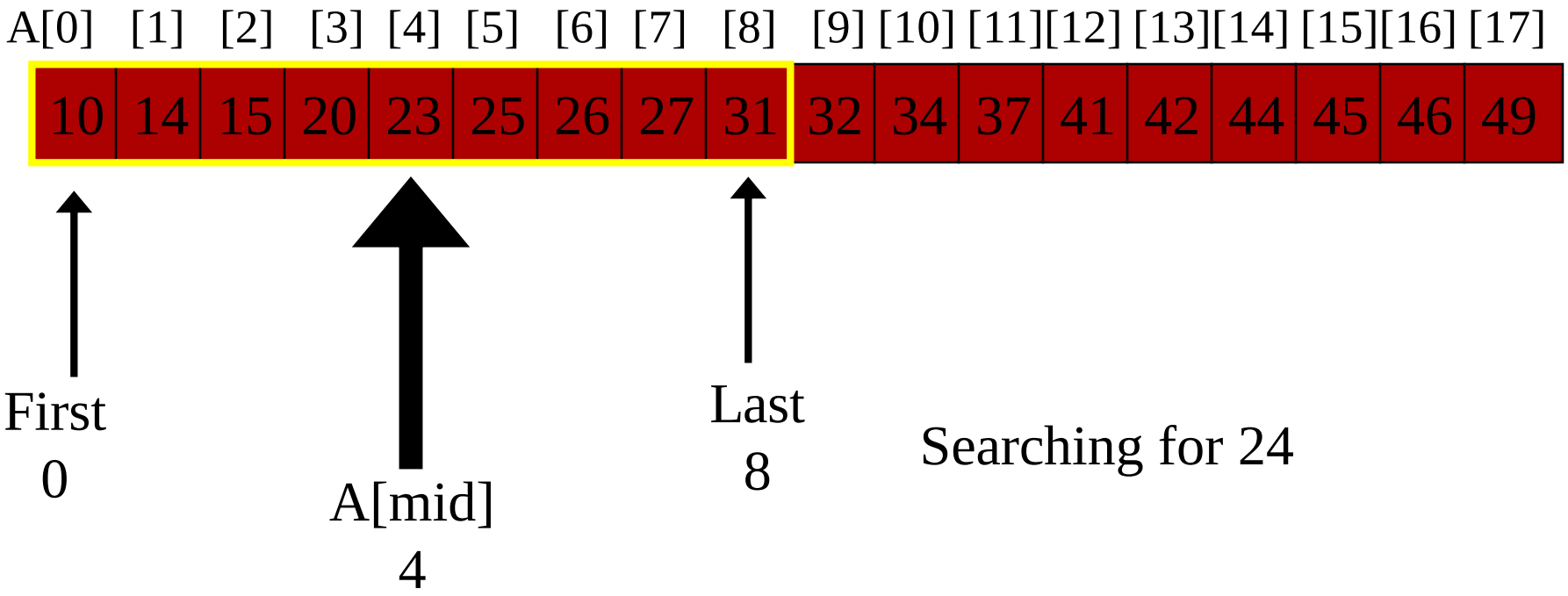
Problem

- **This code works fine for finding an item that is contained in the list.**
- **However, it does not work so well if the item is not in the list.**
- **We get an infinite loop in that case.**

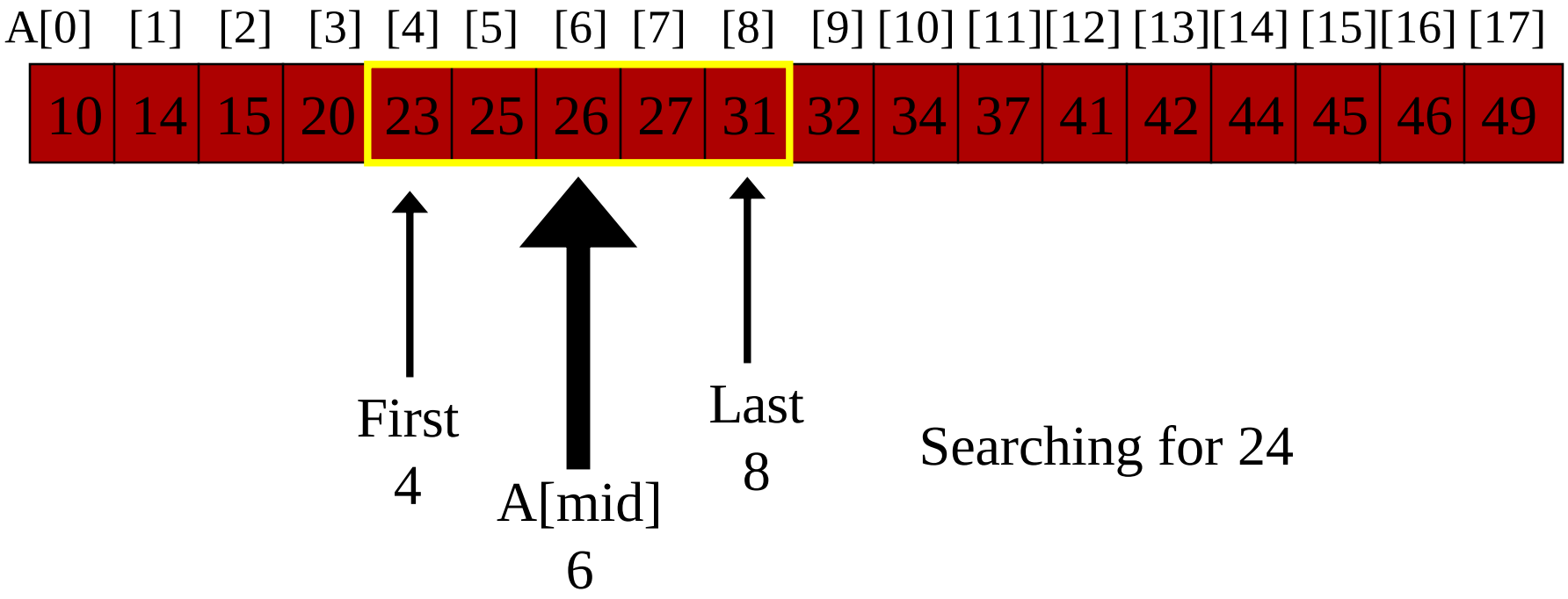
Binary search strategy



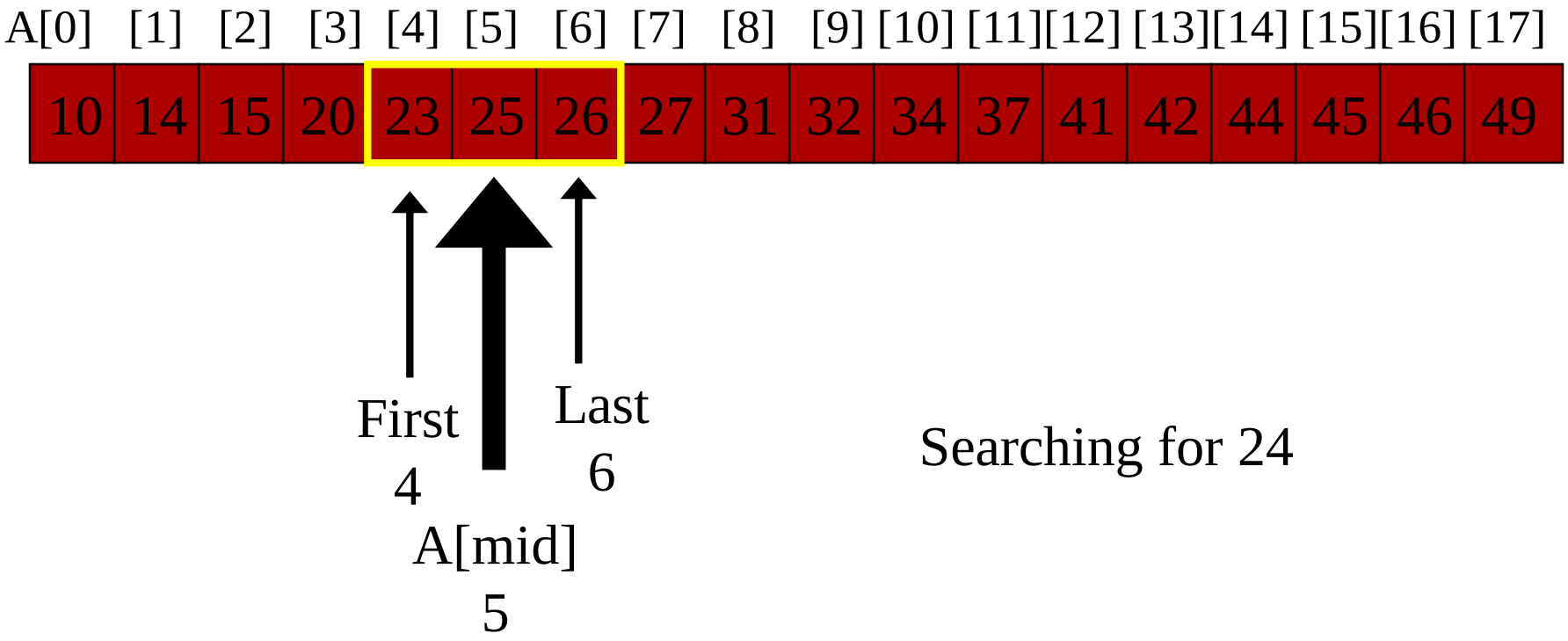
Binary search strategy



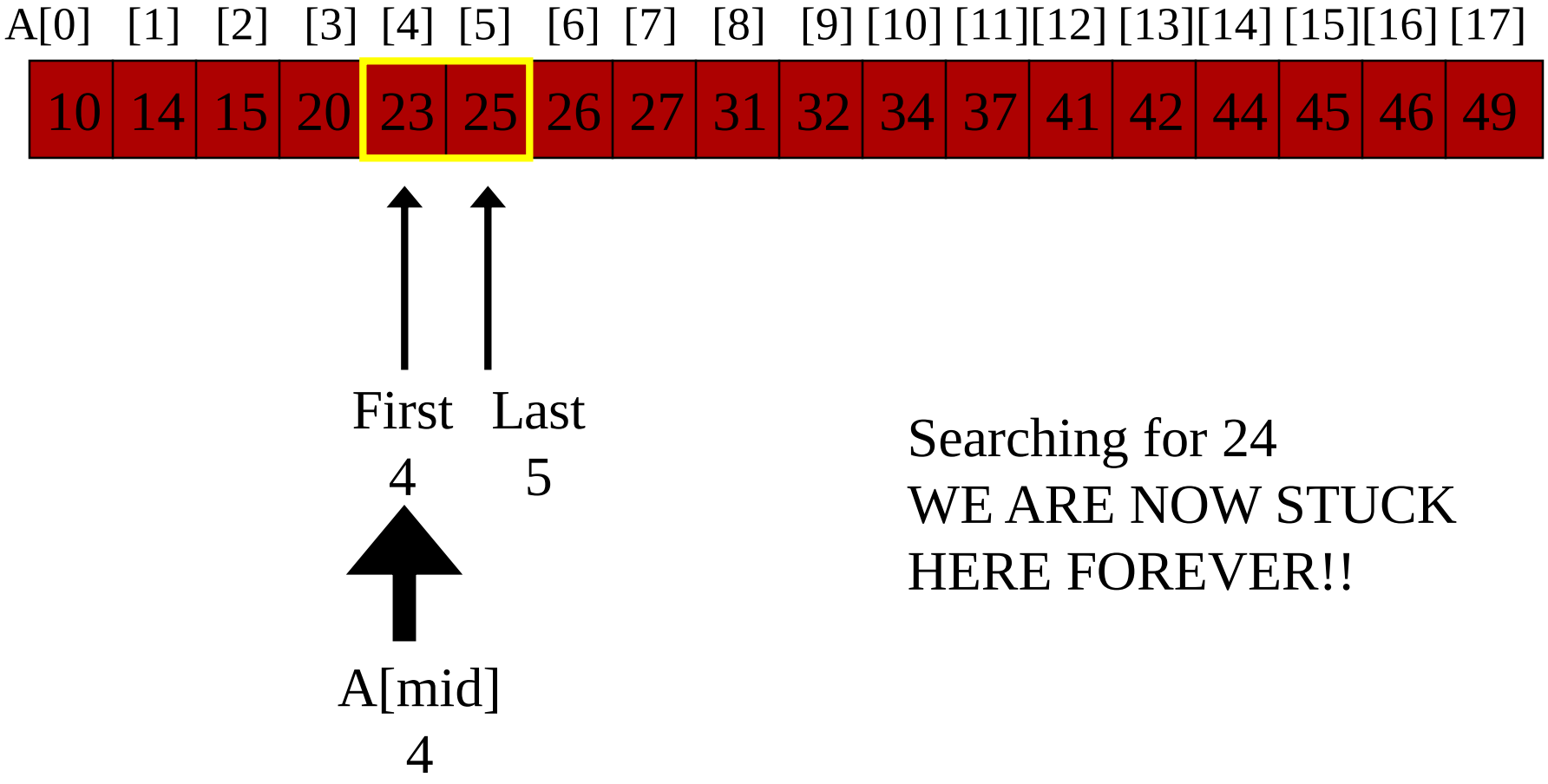
Binary search strategy



Binary search strategy



Binary search strategy



Searching for 24
WE ARE NOW STUCK HERE FOREVER!!

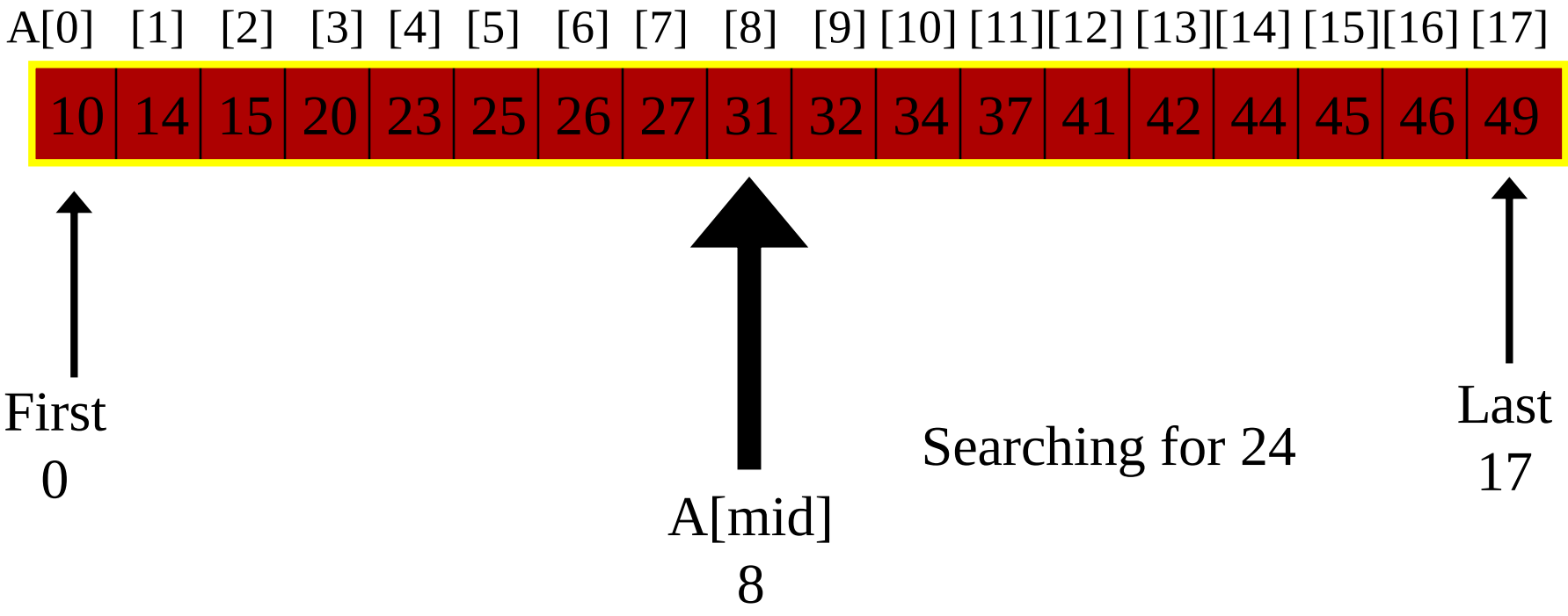
Trace of Code Example 5-2

pass	first	last	mid
1	0	4	2
2	0	2	1
3	1	2	1
4	1	2	1

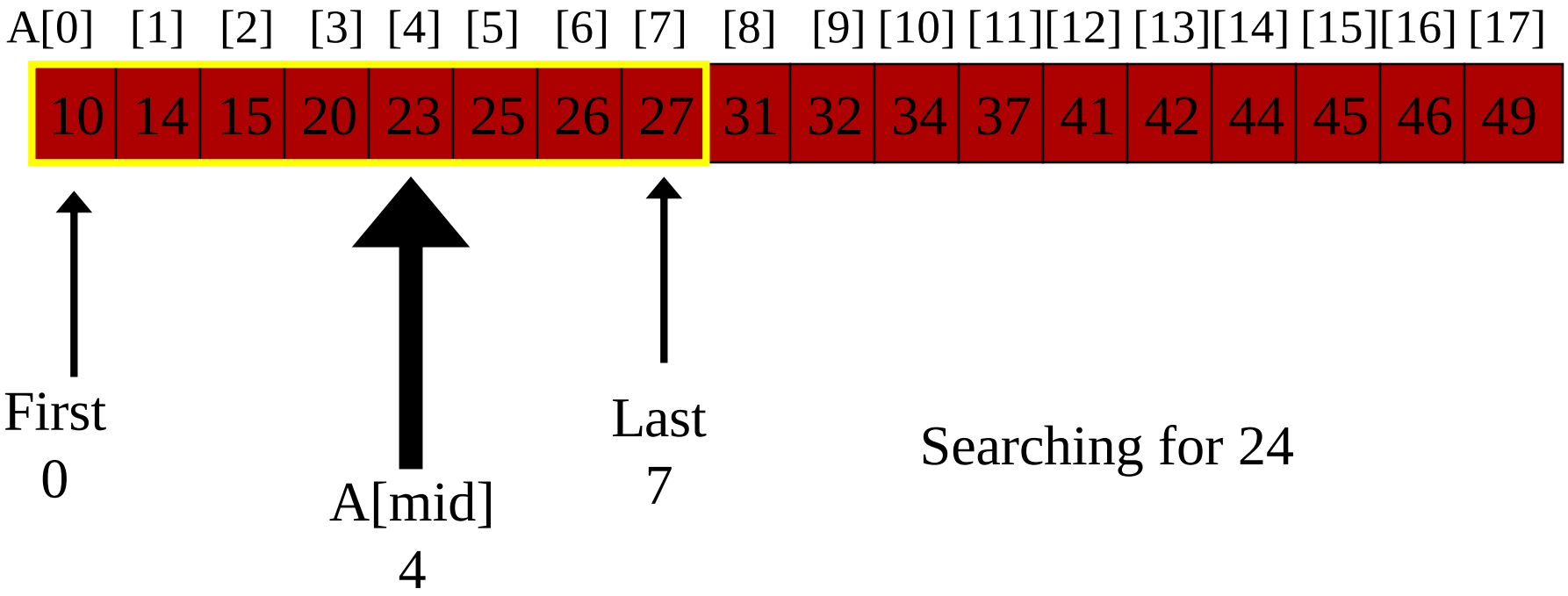
Verified Binary Search

```
int binarySearch(int a[], int n, int target)
{
   // Precondition: array a is sorted in ascending order
   // from a[0] to a[n-1]
   int first(0); int last(n - 1);  int mid;
   while (first <= last) {
mid = (first + last)/2;
 if (target == a[mid])
         return mid;
      else if (target < a[mid])
         last = mid - 1;
      else // must be that target > a[mid]
         first = mid + 1;
   }
   return -1; //use -1 to indicate item not found
}
```

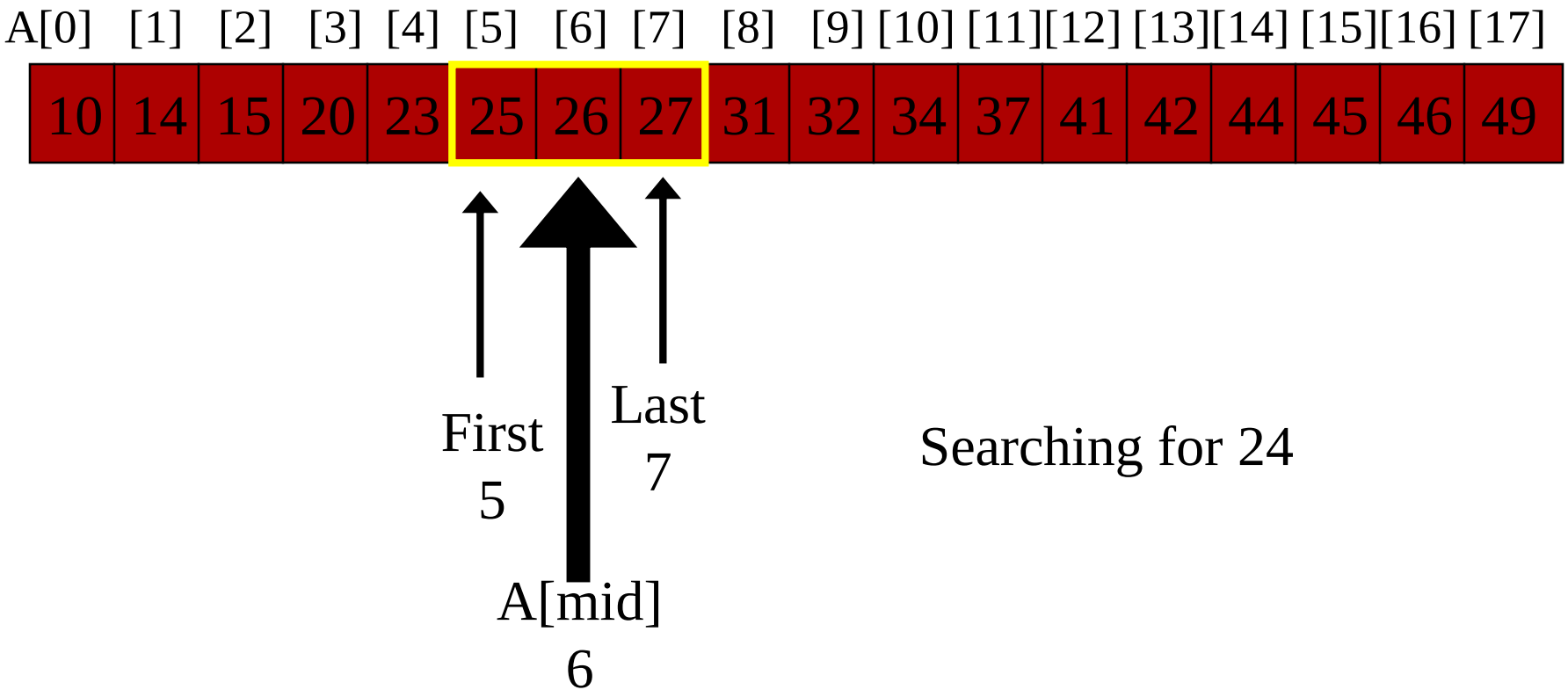
Binary search strategy



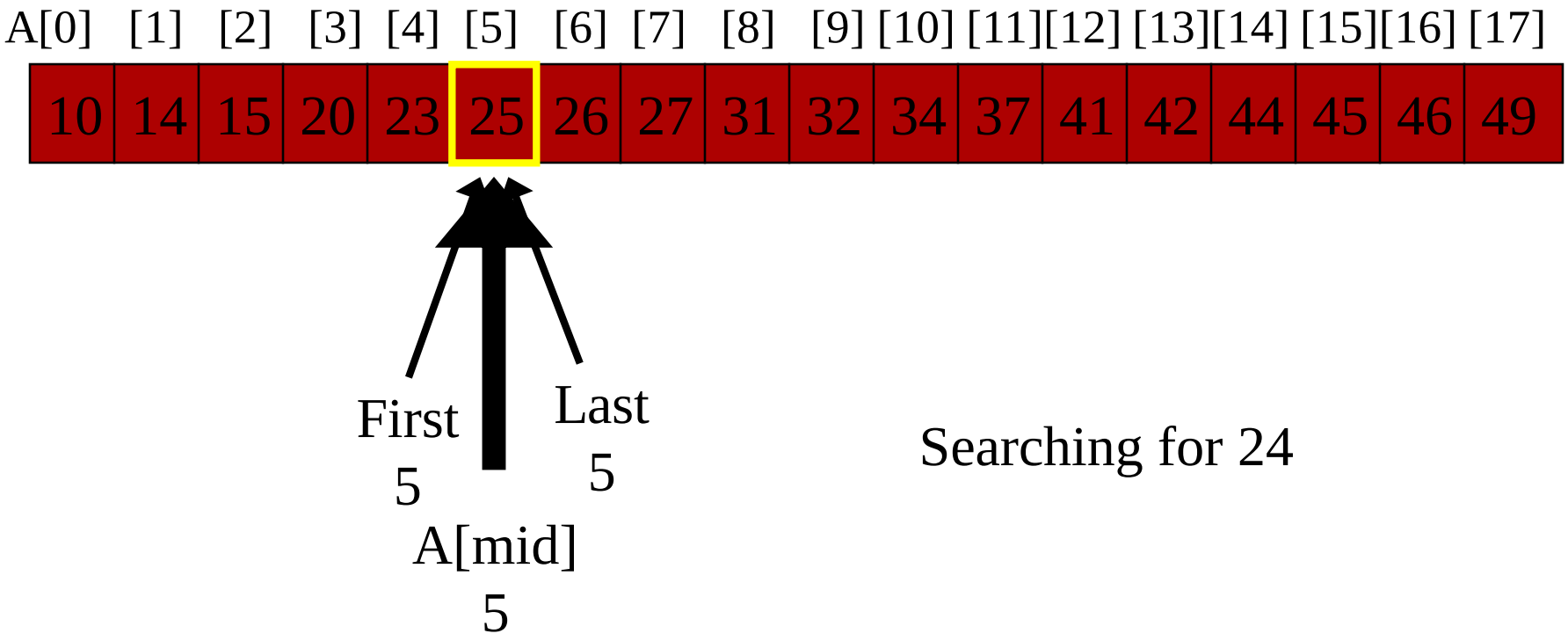
Binary search strategy



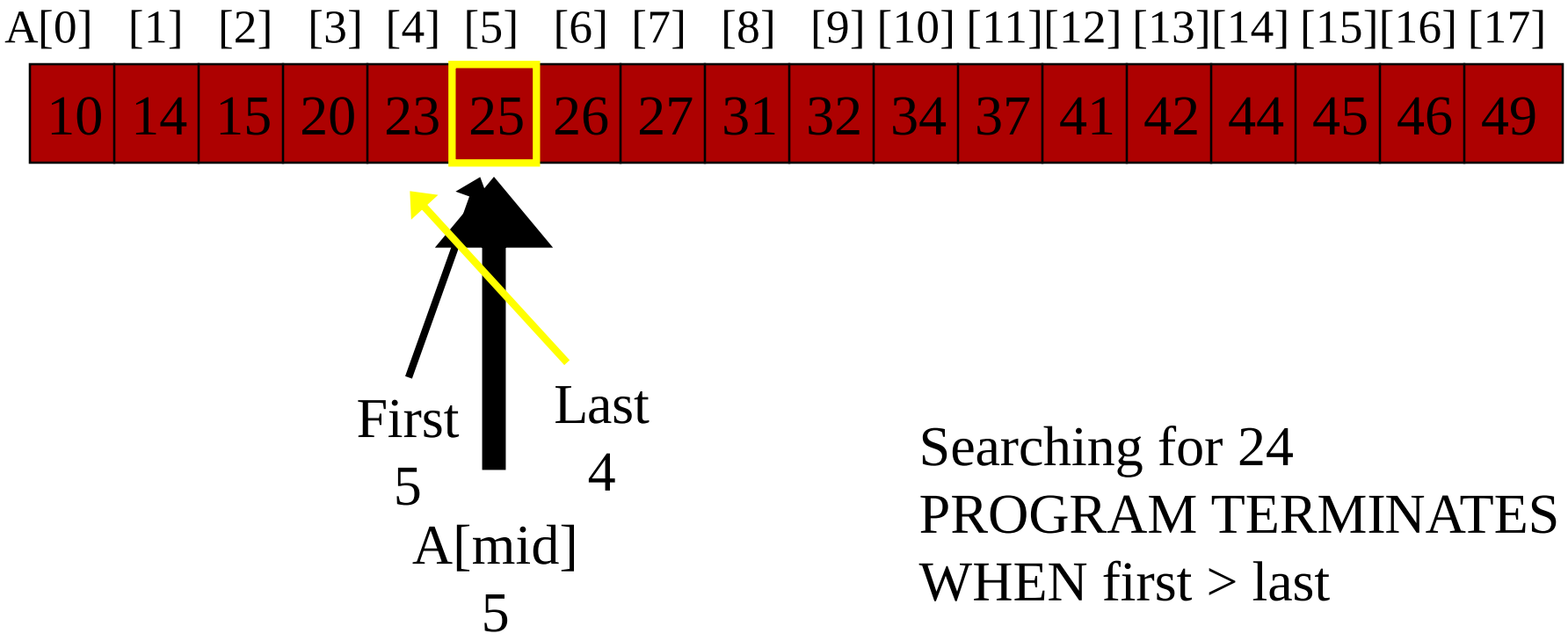
Binary search strategy



Binary search strategy



Binary search strategy



Searching for 24
PROGRAM TERMINATES
WHEN first > last

Retrieval by key

- **A common programming task is to look up an item in a list**
- **We have already seen some simple ways of doing this**
  - Linear search
  - Binary search
- **They differ in terms of data requirements and search efficiency**

Hashing

Hashing is the process of turning the key value into an integer pointer, used to locate a storage location in a larger array.

Hash codes should be designed to give different codes for different keys, although, this cannot be guaranteed.

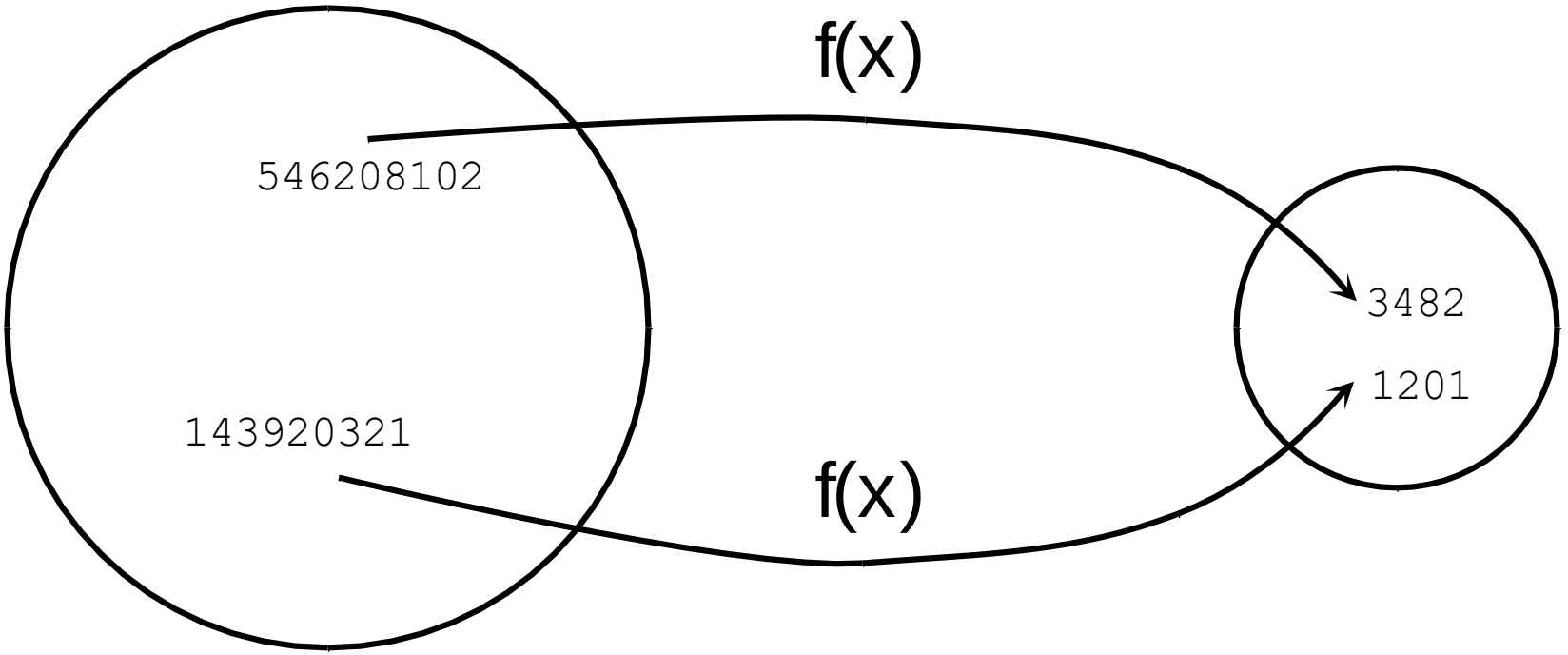
Hashing

- Hashing is a technique that enables searching the data with a complexity **O(1)**.

- Using a key (index) value for each a data record, the key value can be used to represent an array index that is used for data insertion (no sorting time) and retrieval (no search time).

- The challenge in this design is to define a mapping function between the stored record data and the key/index value. Such a mapping function is called hash function.

Hash Function Map



Domain: 1 ... 1,000,000,000

Range: 0 ... 9999

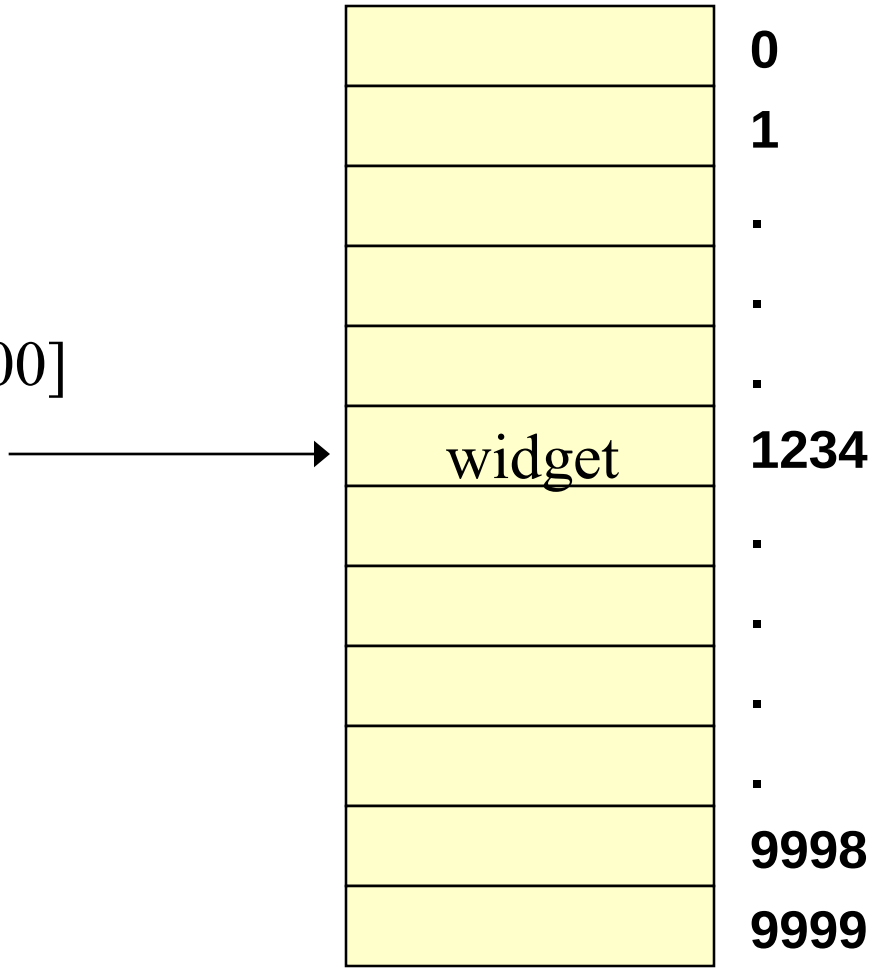
An example of hash conversion

- **You wish to store product information by product number. The product numbers have 5 digits with the lowest one being 10000.**
- **const int max_size = 10000;**
- **Then we could come up with a simple hash function**
- **hashKey = productCode – max_Size;**
- **This gives us a number between 0 and 9999.**
- **We can use this unique number to directly access the array element containing data for that product.**

Product code hash example

Product: Widget
Product code: 11234

ProductArray[11234 - 10000]

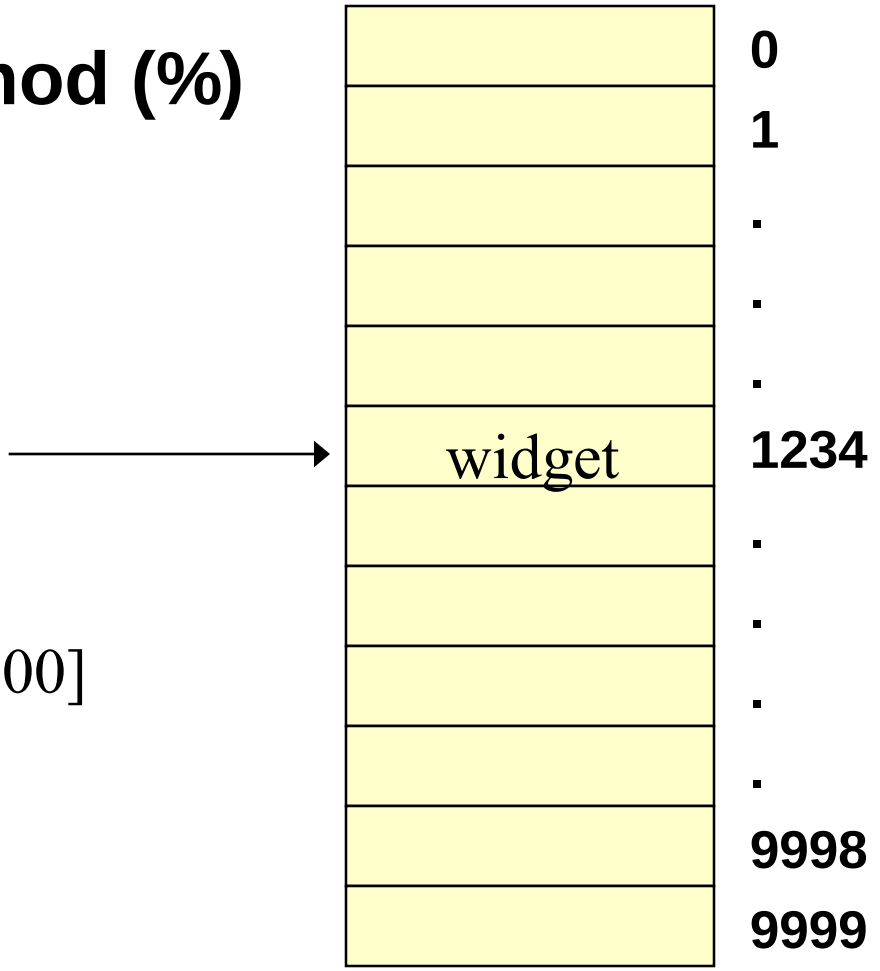


Another implementation

- **Or, we could use mod (%)**
- **like this:**

Product: Widget
Product code: 11234

ProductArray[11234 % 10000]



An internet example

- Internet Protocol (IP) uses 32-bit addresses to look up host names.
- Example: 63.100.1.17 (4 bytes)
- When you want to access a host machine that is not on your network the router takes the host name you give it and looks up the IP address.
- It then forwards your request to that host computer.

Problem

- Routers need to look up addresses as quickly as possible.
- There are millions of IP addresses, so how will it do this?
- Linear search? O(n)
- Binary search? $O(\log_2 n)$
- Neither of these are fast enough.

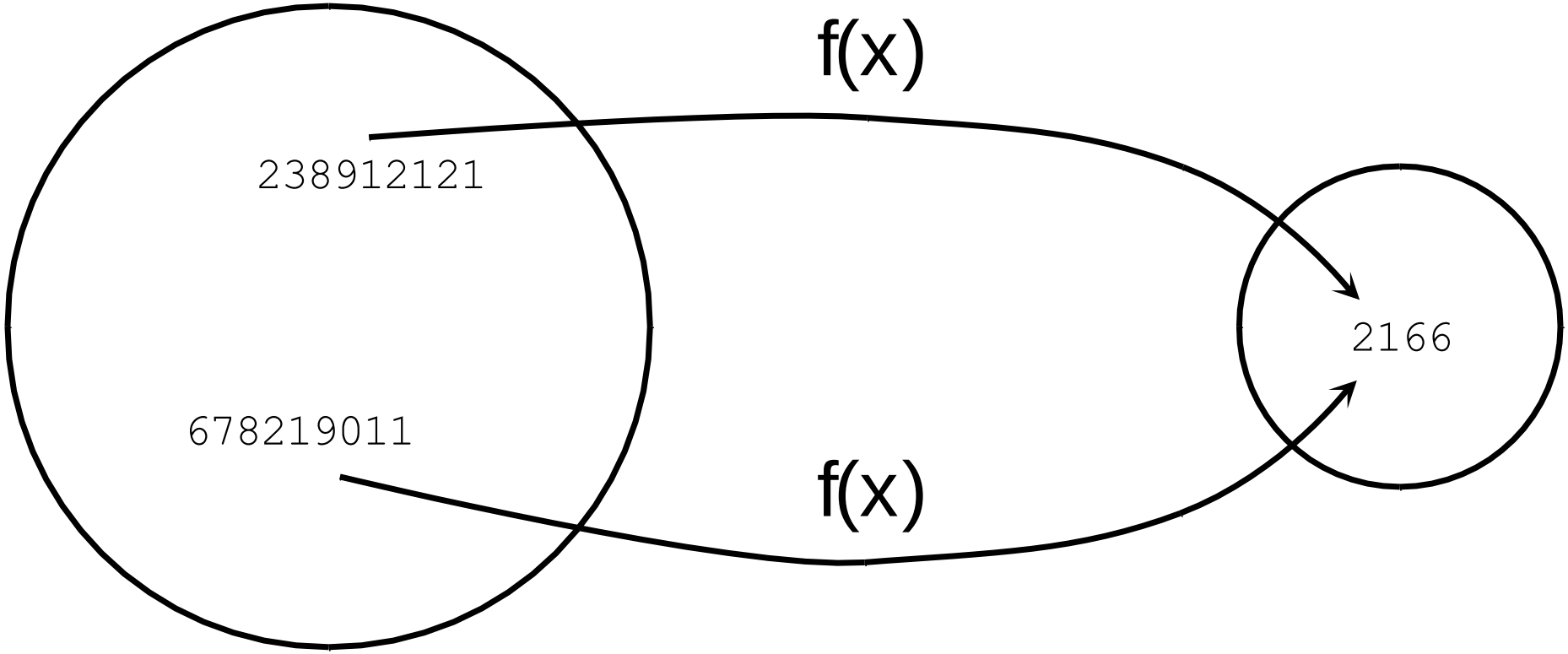
Answer

- **Convert the host name “www.cnn.com” to it’s IP address using a hash function based on the characters.**
- **Perhaps we could use ASCII codes to generate unique numbers within a certain range.**

Issues in hashing

- **Each hash should generate a unique number. If two different items produce the same hash code we have a collision in the data structure. Then what?**
- **Two issues must be addressed**
  1. Hash functions must minimize collisions (there are strategies to do this)
  2. If (when) collisions do occur, we must know how to handle them.

A Collision

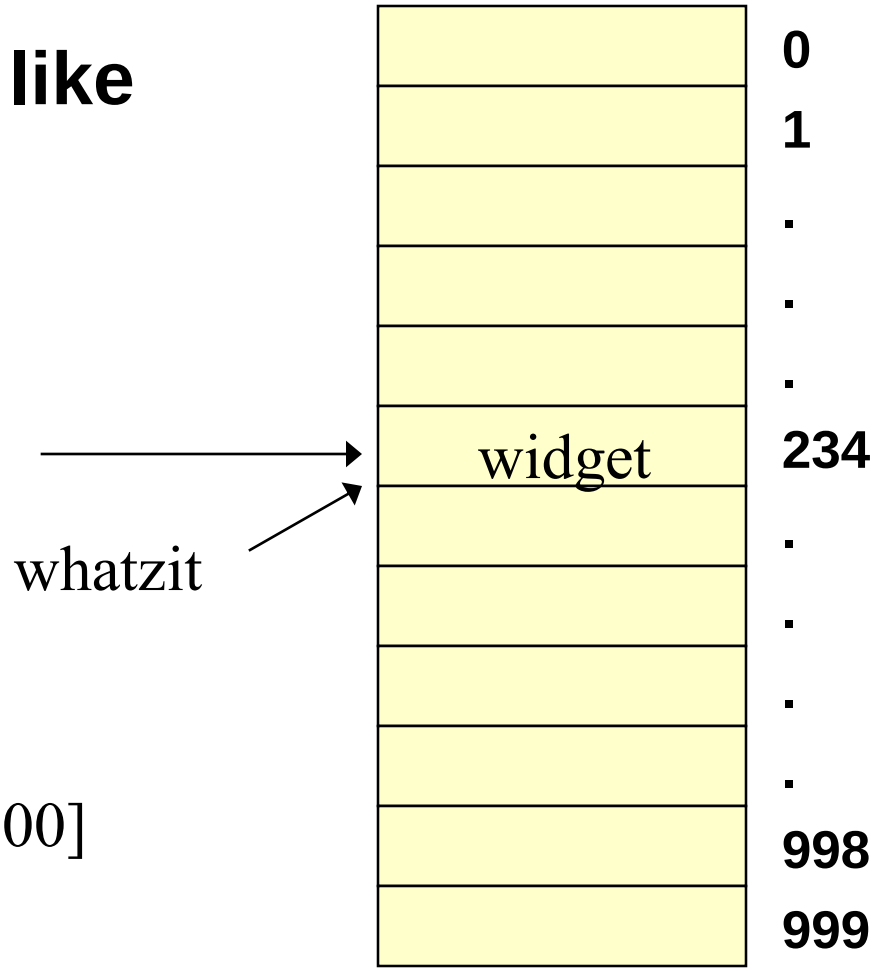


What if...

- **We used mod (%) like**
- **this:**

Product: Widget
Product code: 11234
Product: Whatzit
Product code: 12234

ProductArray[11234 % 1000]



Two good rules to follow

- **A good hashing function must**
  - 1. Always produce the same address for a given key, and
  - 2. For two different keys, the probability of producing the same address must be low.

Collision

- Same hash value for two different records.
- Generally, a collision-free hash function is not easy to design.
  - A usual approach is to use an array size that is larger than needed. The extra array positions make the collisions less likely
  - A good hash function will distribute the keys uniformly throughout the locations of the array.

Collison Resolution

- One way to resolve collisions is to place the colliding record in another location that is still open.
  - Open-addressing collision resolution techniques
    - Linear probing
    - double hashing
- Chained hashing collision resolution techniques

Open Addressing

- Open addressing requires that the array be initialized so that the program can test if an array position already contains a record.
  - A mechanism is required to determine the next array index to be searched.
    - Linear Probing
    - Double Hashing

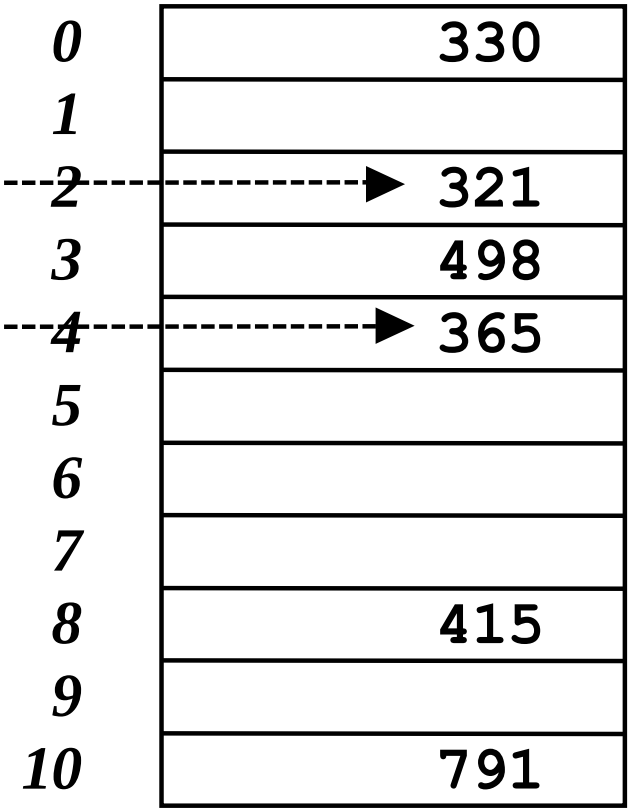
Linear Probing

- if hash(x) is full check for [hash(x) + 1].
  - A common problem with linear probing is ***clustering***
  - Clustering makes insertions take longer because the insert function must step all the way through a cluster to find a vacant location. Searches require more time for the same reason.

Hash table after 365 added

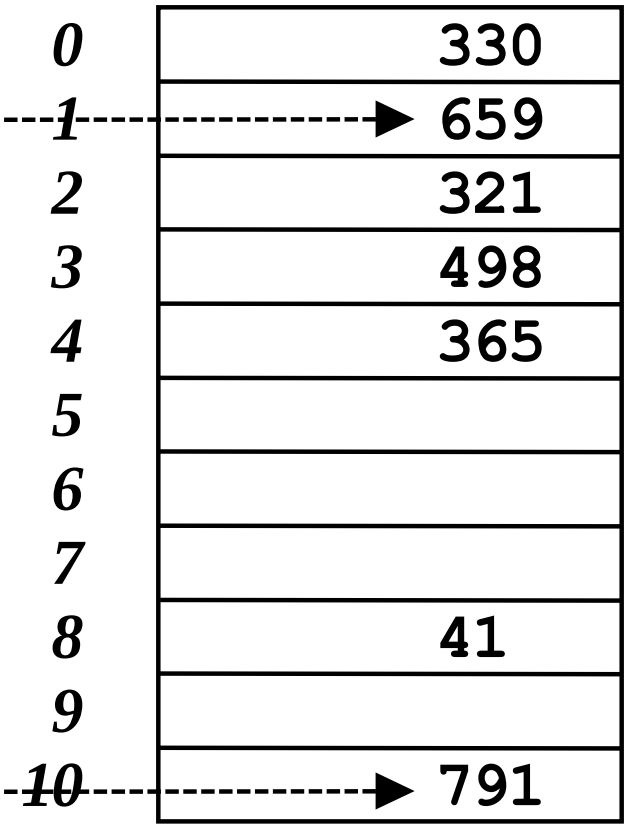
1. can't add 365 at its home address

2. so probe forward, adding it at the next free slot



Hash table after 659 added

2. wrapping around, next free space found here



1. can't add 659 at its home address

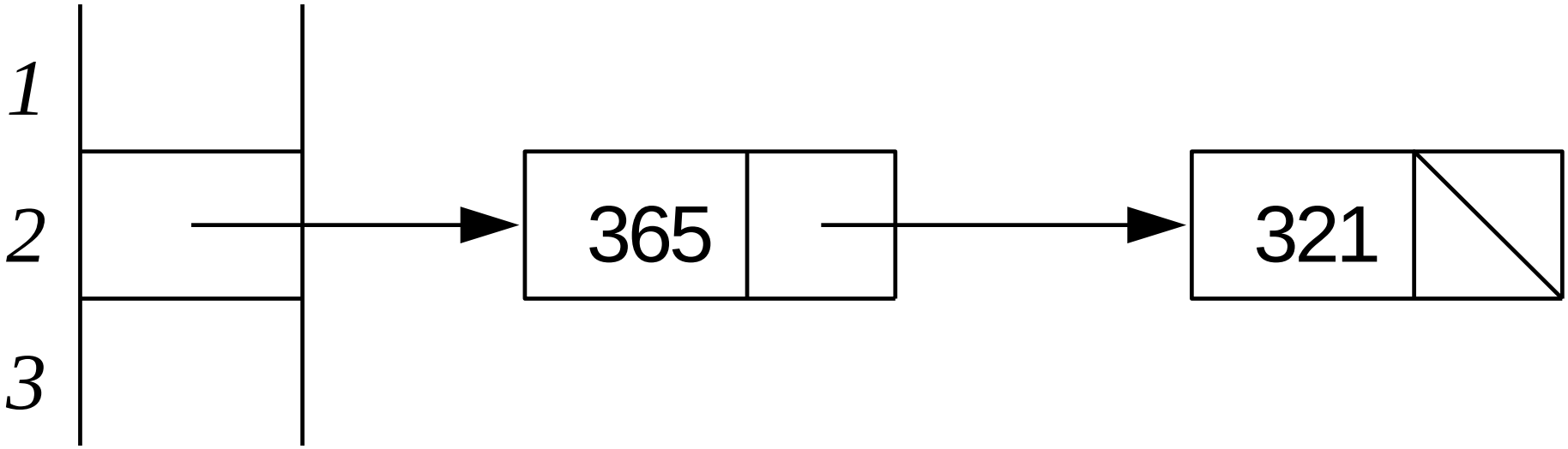
Double Hashing

- Double Hashing is the most common technique to avoid clustering.
- Let *i* = *hash(key)*.
  - If the location data[i] already contains a record then let *i* = *(i+hash2(key))* and try the new *data[i]*.
  - If that location already contains a record, then let *i* = *(i+hash2(key))*, and try that *data[i]*, and so forth until a vacant position is found.

Chained Hashing

- In open addressing, each array element can hold just one entry. When the array is full, no more records can be added to the table.
- A better approach is called chained hashing
  - Each component of the hash table's array can hold more than one entry.
- We still hash the key of each entry, but upon collision, we simply place the new entry in its proper array component along with other entries that happened to hash to the same array index.
- The most common way to implement chaining is to have each array element be a linked list. The nodes in a particular linked list will each have a key that hashes to the same value.

Chained Hashing



Hash table class definition

```
template < class tableKeyType, class tableDataType >
class Table
{ public:
   Table(); // Table constructor
   void insert(const tableKeyType & key,
                     const tableDataType & data);
   // Precondition: None
   // Postcondition: data and key are stored in the Table,
   //   i.e., lookup(key,data) == true and returns the data
   //   Note: If data already stored with this key, the insert
   //         call will replace it.
```

Public section (continued)

```
bool lookup(const tableKeyType & key,
                        tableDataType & data);
// Precondition: None
// Postcondition: if key in table, returns true and associated
//    data is returned; otherwise, false is returned and
//    data is undefined.
void deleteKey(const tableKeyType & key);
// Precondition: None
// Postcondition: lookup(key,d) returns false
void dump(); // print the contents of the hash table
```

Private section

```
private:
  // implementation via a hash table
  struct Slot;
  typedef Slot * Link;
  struct Slot {
     tableKeyType key;
     tableDataType data;
     Link next;
  };
  Link T[MAX_TABLE]; // table is an array of pointers to slots
```

Private section (continued)

```
int hash(const tableKeyType & key);
   // Precondition: none
   // Postcondition: none
   // Returns: the home address for key
bool search(Link & slotPointer, const tableKeyType & target);
   // Precondition: slotPointer points to the start of the chain
   // for target's hash address
   // Postcondition: if target is in the chain, slotPointer points
to it
   // Returns: true if target is in the table, else false
};
```

hash() implementation

```
template < class tableKeyType, class tableDataType >
int
Table < tableKeyType, tableDataType >
::hash(const tableKeyType & key)
{
   return key % MAX_TABLE;
}
```

Search() implementation

```
template < class tableKeyType, class tableDataType >
bool Table < tableKeyType, tableDataType >
::search(Link & slotPointer, const tableKeyType & target)
{
  // search for target, starting at slotPointer
  for ( ; slotPointer; slotPointer = slotPointer -> next)
    if (slotPointer->key == target)
      return true;
  return false;
}
```

Table constructor

```
template < class tableKeyType, class tableDataType >
Table < tableKeyType, tableDataType >
::Table() // implementation of Table constructor
{  int i;
   for (i = 0; i < MAX_TABLE; i++)
      T[i] = 0;
}
```

Insert() implementation

```
template < class tableKeyType, class tableDataType >
void Table < tableKeyType, tableDataType >
::insert(const tableKeyType & key, const tableDataType
& data)
{  int pos(hash(key)); // find a position to insert the item
   Link sp(T[pos]);
```

Insert() continued

```
if (!search(sp, key)) { // key was not in the table
    // insert new item at beginning of list
    Link insertedSlot = new Slot;
    insertedSlot->key = key;
    insertedSlot->data = data;
    insertedSlot->next = T[pos];
    T[pos] = insertedSlot;
  }
  else  // found old record -- update the data
    sp->data = data;
}
```

lookup() implementation

```
template < class tableKeyType, class tableDataType >
bool Table < tableKeyType, tableDataType >
::lookup(const tableKeyType & key, tableDataType &
data)
{  int pos(hash(key));
   Link sp(T[pos]);
   if (search(sp, key)) {
      data = sp->data;
      return true; }
   else
      return false;
}
```

deleteKey() implementation

```
template < class tableKeyType, class tableDataType >
void
Table < tableKeyType, tableDataType >::deleteKey(const
tableKeyType & key)
{  // need to find pointer to item preceding the slot to delete
   int pos(hash(key));
   Link p;
   if (T[pos] == 0) return; // there's no list for this slot
   if (T[pos]->key == key) { // special case, first item in chain
      Link deleteSlot(T[pos]);
      T[pos] = T[pos]->next;
      delete deleteSlot; }
```

deleteKey() (continued)

```
    else
      for (p = T[pos]; p->next; p = p->next)
        if (p->next->key == key) {
          Link deleteSlot = p->next;
          p->next = p->next->next;
          delete deleteSlot;
          break;
        }
}
```

dump() implementation

```
template < class tableKeyType, class tableDataType >
void Table < tableKeyType, tableDataType >::dump()
{  int i;
   for (i = 0; i < MAX_TABLE; i++) {
      cout << i << '\t';
      Link p;
      for (p = T[i]; p; p = p->next)
         cout << p->key << '\t';
      cout << '\n';
   }
    cout << '\n';
}
```

Search Methods Summary

- Recall what we know about searching:

Method	Time	Drawbacks
Sequential	O(n)	Slow for large n
Binary	$O(\log_2 n)$	Data must be contiguous Inserts and deletes are slow Data must be sorted first
Tree	$O(\log_2 n)$	Tree must be balanced
Hashing	O(1)	Needs much unused memory
Direct access	O(1)	Keys must match array indices