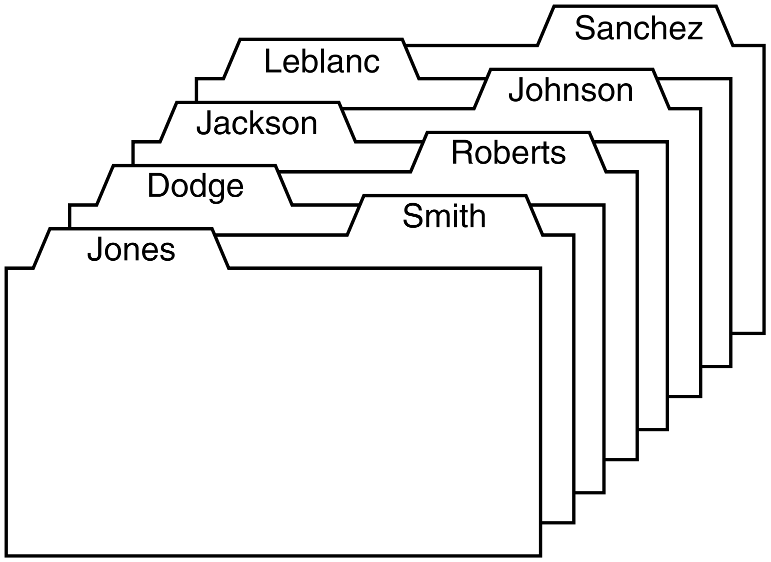
Chapter 7

SEARCHING

Records and Keys



- We are given a ***list*** of ***records***, where each record is associated with one piece of information, which we shall call a ***key***.
- We are given one key, called the ***target***, and are asked to search the list to find the record(s) (if any) whose key is the same as the target.
- There may be more than one record with the same key, or there may be no record with a given key.
- We often ask how times one key is compared with another during a search. This gives us a good measure of the total amount of work that the algorithm will do.
- ***Internal searching*** means that all the records are kept in high-speed memory. In ***external searching***, most of the records are kept in disk files. We study only internal searching.
- For now, we consider only contiguous lists of records. We shall consider linked structures in Chapter 10.

Records and Keys in C++

The records (from a **class** Record) that are stored in the list being searched (generally called the_list) must conform to the following minimal standards:

- Every Record is associated to a key (of a type or **class** called Key).
- Key objects can be compared with the standard operators

  == , != , <, >, <= , >= .

- There is a conversion operation to turn any Record into its associated Key.
- Record objects can be compared to each other or to a Key by first converting the Record objects to their associated keys.
- Examples of the conversion operation:
  - A method of the **class** Record, with the declaration **operator** Key( ) **const;**
  - A constructor for the class Key, with declaration Key(**const** Record &);
  - If the classes Key and Record are identical, no conversion needs to be defined, since any Record is automatically a Key.

- We do not assume that a Record has a Key object as a data member, although often it does. We merely assume that the compiler can turn a Record into its corresponding Key.

Parameters for Search Functions

Each searching function has two input parameters:

- First is the *list* to be searched;
- Second is the *target* key for which we are searching.

Each searching function will also have an output parameter and a returned value:

- The returned value has type Error_code and indicates whether or not the search is successful in finding an entry with the target key.
- If the search is successful, then the returned value is success, and the output parameter called position will locate the target within the list.
- If the search is unsuccessful, then the value not_present is returned, and the output parameter may have an undefined value or a value that will differ from one method to another.

Key Definition in C++

To select existing types as records and keys, a client could use type definition statements such as:

```
typedef int Key;
typedef int Record;
```

A client can design new classes that display appropriate behavior based on the following skeletons:

```
//    Definition of a Key class:
  class Key{
    public:
      //    Add any constructors and methods for key data.
    private:
      //    Add declaration of key data members here.
  };
//    Declare overloaded comparison operators for keys:
  bool operator ==  (const Key &x, const Key &y);
  bool operator >   (const Key &x, const Key &y);
  bool operator <   (const Key &x, const Key &y);
  bool operator >=  (const Key &x, const Key &y);
  bool operator <=  (const Key &x, const Key &y);
  bool operator !=  (const Key &x, const Key &y);
//    Definition of a Record class:
  class Record{
    public:
      operator Key( );            //    implicit conversion from Record to Key.
     //    Add any constructors and methods for Record objects.
    private:
     //    Add data components.
  };
```

Sequential Search and Analysis

Begin at one end of the list and scan down it until the desired key is found or the other end is reached:

```
Error_code sequential_search(const List<Record> &the_list,
                             const Key &target, int &position)
/* Post: If an entry in the_list has key equal to target, then return success and the
         output parameter position locates such an entry within the list.
         Otherwise return not_present and position becomes invalid. */
{
   int s = the_list.size( );
   for (position = 0; position < s; position++) {
      Record data;
      the_list.retrieve(position, data);
      if (data == target) return success;
   }
   return not_present;
}
```

To analyze the behavior of an algorithm that makes comparisons of keys, we shall use the count of these key comparisons as our measure of the work done.

The number of comparisons of keys done in sequential search of a list of length n is

- Unsuccessful search: n comparisons.
- Successful search, best case: 1 comparison.
- Successful search, worst case: n comparisons.
- Successful search, average case: $\frac{1}{2}(n+1)$ comparisons.

Testing Program

- For test purposes, use integer keys, and do not store any data other than a key in a record. Accordingly, we define **typedef** Key Record;
- Keep a count of all key comparison operations, by modifying the overloaded key comparison operations to increment a counter whenever they are called.
- This counter must be available to all Key objects: Thus, it should be declared as a **static** class member. In C++, static class members provide data objects that are shared by every instance of the class.

```
class Key {
   int key;
public:
   static int comparisons;
   Key (int x = 0);
   int the_key( ) const;
};

bool operator ==  (const Key &x, const Key &y);
bool operator >   (const Key &x, const Key &y);
bool operator <   (const Key &x, const Key &y);
bool operator >=  (const Key &x, const Key &y);
bool operator <=  (const Key &x, const Key &y);
bool operator !=  (const Key &x, const Key &y);
```

- The method the_key inspects a copy of a key's value.
- The static counter comparisons is incremented by any call to a Key comparison operator. For example:

```
bool operator == (const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}
```

- Static data members must be defined and initialized outside of a class definition. Accordingly, the following statement is included in the Key implementation file `key.c`:

```
int Key::comparisons = 0;
```

- Use the **class** Timer from Appendix C to provide CPU timing information.

Test Data for Searching

- Most later searching methods require the data to be ordered, so use a list with integer keys in increasing order.
- To test both successful and unsuccessful searches, insert only keys containing odd integers into the list.
- If n denotes the number of entries in the list, then the targets for successful searches will be

$$1, 3, 5, \ldots, 2n-1.$$

- For unsuccessful searches, the targets will be

$$0, 2, 4, 6, \ldots, 2n.$$

- In this way we test all possible failures, including targets less than the smallest key in the list, between each pair, and greater than the largest.
- To make the test more realistic, use pseudo-random numbers to choose the target, by employing the method

Random :: random_integer

from Appendix B.

Ordered Lists

DEFINITION An ***ordered list*** is a list in which each entry contains a key, such that the keys are in order. That is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j.

- All List operations except insert and replace apply without modification to an ordered list.
- Methods insert and replace must fail when they would otherwise disturb the order of a list.
- We implement an ordered list as a class *derived* from a contiguous List. In this derived class, we shall override the methods insert and replace with new implementations.

```
class Ordered_list: public List<Record>{
public:
   Ordered_list( );
   Error_code insert(const Record &data);
   Error_code insert(int position, const Record &data);
   Error_code replace(int position, const Record &data);
};
```

Overloaded Insertion

- We also overload the method insert so that it can be used with a single parameter. The position is automatically determined so as to keep the resulting list ordered:

```
Error_code Ordered_list :: insert(const Record &data)
/* Post: If the Ordered_list is not full, the function succeeds: The Record data
         is inserted into the list, following the last entry of the list with a strictly lesser
         key (or in the first list position if no list element has a lesser key).
         Else: the function fails with the diagnostic Error_code overflow. */
{
   int s = size( );
   int position;
   for (position = 0; position < s; position++) {
      Record list_data;
      retrieve(position, list_data);
      if (data >= list_data) break;
   }
   return List<Record> :: insert(position, data);
}
```

Overridden Insertion

- The scope resolution is necessary, because we have overridden the original List insert with a new Ordered_list method:

```
Error_code Ordered_list :: insert(int position, const Record &data)
/* Post: If the Ordered_list is not full, 0 ≤ position ≤ n, where n is the
         number of entries in the list, and the Record data can be inserted at
         position in the list, without disturbing the list order, then the function
         succeeds: Any entry formerly in position and all later entries have their
         position numbers increased by 1 and data is inserted at position of the
         List.
         Else: the function fails with a diagnostic Error_code. */
{
   Record list_data;
   if (position > 0) {
      retrieve(position − 1, list_data);
      if (data < list_data)
         return fail;
   }
   if (position < size( )) {
      retrieve(position, list_data);
      if (data > list_data)
         return fail;
   }
   return List<Record> :: insert(position, data);
}
```

- Note: The *overridden* method replaces a method of the base class by one with a matching name and parameter list. The *overloaded* method matches in name but has a different parameter list.

Binary Search

- **Idea:** In searching an ordered list, first compare the target to the key in the center of the list. If it is smaller, restrict the search to the left half; otherwise restrict the search to the right half, and repeat. In this way, at each step we reduce the length of the list to be searched by half.
- **Keep** two indices, top and bottom, that will bracket the part of the list still to be searched.

> The target key, provided it is present, will be found between the indices bottom and top, inclusive.

- **Initialization:** Set bottom = 0; top = the_list.size( ) − 1;
- **Compare** target with the Record at the midpoint,

  mid = (bottom + top)/2;

- **Change** the appropriate index top or bottom to restrict the search to the appropriate half of the list.
- **Loop terminates** when top $\le$ bottom, if it has not terminated earlier by finding the target.
- **Make progress** toward termination by ensuring that the number of items remaining to be searched, top − bottom + 1, strictly decreases at each iteration of the process.

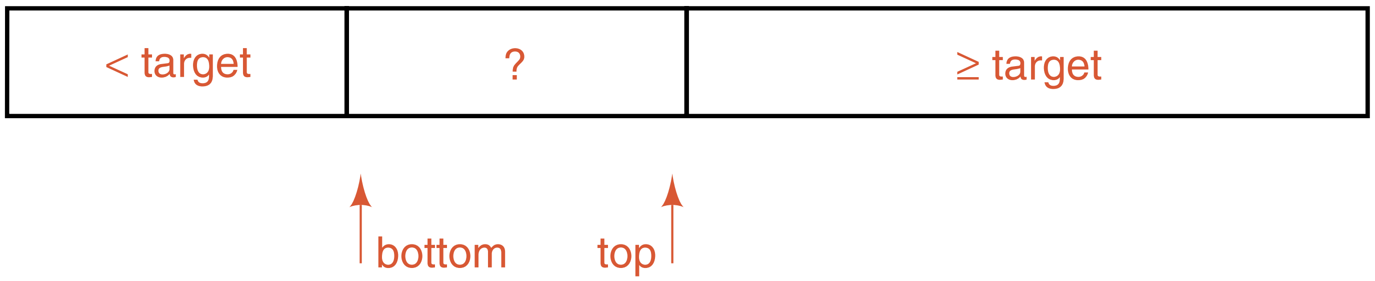
The Forgetful Version of Binary Search

- **Method:** Forget the possibility that the Key target might be found quickly and continue, whether target has been found or not, to subdivide the list until what remains has length 1.

```
Error_code recursive_binary_1(const Ordered_list &the_list,
        const Key &target, int bottom, int top, int &position)
/* Pre:  bottom and top define the range in the list to search for the target.
   Post: If a Record in the range of locations from bottom to top in the_list has
         key equal to target, then position locates one such entry and a code
         of success is returned. Otherwise, the Error_code of not_present is
         returned and position becomes undefined.
   Uses: recursive_binary_1 and methods of the classes List and Record.  */
{  Record data;
   if (bottom < top) {           //  List has more than one entry.
      int mid = (bottom + top)/2;
      the_list.retrieve(mid, data);
      if (data < target)         //  Reduce to top half of list.
         return recursive_binary_1(the_list, target,
                                   mid + 1, top, position);
      else                       //  Reduce to bottom half of list.
         return recursive_binary_1(the_list, target,
                                   bottom, mid, position);
   }
   else if (top < bottom)
      return not_present;        //  List is empty.
   else {                        //  List has exactly one entry.
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data ==  target) return success;
      else return not_present;
   }
}
```

Algorithm Verification

- The division of the list into sublists is described in the following diagram:



- Only entries *strictly* less than target appear in the first part of the list, but the last part contains entries greater than or *equal* to target.

- When the middle part of the list is reduced to size 1, it will be guaranteed to be the *first* occurrence of the target if it appears more than once in the list.

- We must prove carefully that the search makes progress towards termination. This requires checking the calculations with indices to make sure that the size of the remaining sublist strictly decreases at each iteration. It is also necessary to check that the comparison of keys corresponds to the division into sublists in the above diagram.

Nonrecursive Binary Search

```
Error_code binary_search_1 (const Ordered_list &the_list,
                                     const Key &target, int &position)
/* Post: If a Record in the_list has Key equal to target, then position lo-
         cates one such entry and a code of success is returned. Otherwise,
         not_present is returned and position is undefined.
   Uses: Methods for classes List and Record.  */

{
   Record data;
   int bottom = 0, top = the_list.size( ) − 1;

   while (bottom < top) {
      int mid = (bottom + top)/2;
      the_list.retrieve(mid, data);
      if (data < target)
         bottom = mid + 1;
      else
         top = mid;
   }

   if (top < bottom) return not_present;
   else {
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data ==  target) return success;
      else return not_present;
   }
}
```

Recognizing Equality in Binary Search

- **Method:** Check at each stage to see if the target has been found.

```
Error_code recursive_binary_2(const Ordered_list &the_list,
                const Key &target, int bottom, int top, int &position)
/* Pre:  The indices bottom to top define the range in the list to search for the
         target.
   Post: If a Record in the range from bottom to top in the_list has key equal
         to target, then position locates one such entry, and a code of suc-
         cess is returned. Otherwise, not_present is returned, and position is
         undefined.
   Uses: recursive_binary_2, together with methods from the classes
         Ordered_list and Record. */
{
   Record data;
   if (bottom <= top) {
      int mid = (bottom + top)/2;
      the_list.retrieve(mid, data);
      if (data == target) {
         position = mid;
         return success;
      }
      else if (data < target)
         return recursive_binary_2(the_list, target,
                                   mid + 1, top, position);
      else
         return recursive_binary_2(the_list, target,
                                   bottom, mid − 1, position);
   }
   else return not_present;
}
```

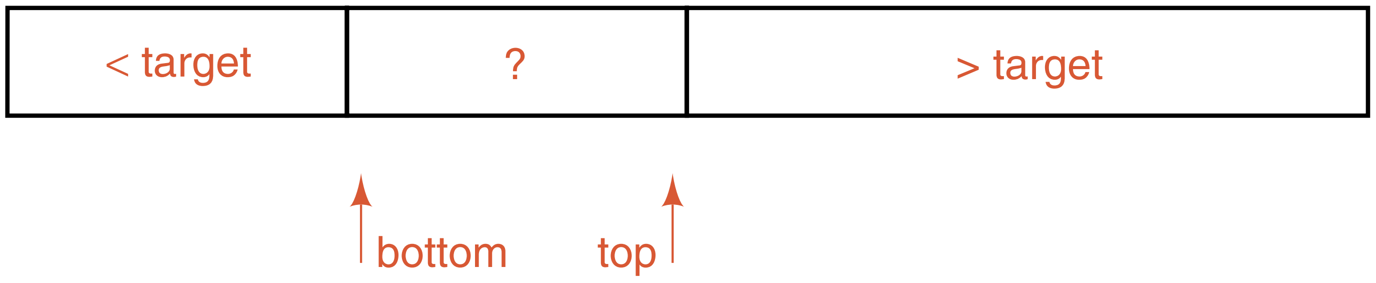
Nonrecursive Version

```
Error_code binary_search_2(const Ordered_list &the_list,
                           const Key &target, int &position)
/* Post: If a Record in the_list has key equal to target, then position lo-
         cates one such entry and a code of success is returned. Otherwise,
         not_present is returned and position is undefined.
   Uses: Methods for classes Ordered_list and Record. */

{
   Record data;
   int bottom = 0, top = the_list.size( ) − 1;
   while (bottom <= top) {
      position = (bottom + top)/2;
      the_list.retrieve(position, data);
      if (data == target) return success;
      if (data < target) bottom = position + 1;
      else top = position − 1;
   }
   return not_present;
}
```

Algorithm Verification

- The division of the list into sublists is described in the following diagram:



- The first part of the list contains only entries strictly less than target, and the last part contains only entries strictly greater than target.

- If target appears more than once in the list, then the search may return any instance of the target.

- Proof of progress toward termination is easier than for the first method.

Comparison Trees: Definitions

The ***comparison tree*** of an algorithm is obtained by tracing the action of the algorithm, representing each comparison of keys by a ***vertex*** of the tree (which we draw as a circle). Inside the circle we put the index of the key against which we are comparing the target key.

Branches (lines) drawn down from the circle represent the possible outcomes of the comparison. When the algorithm terminates, we put either F (for failure) or the location where the target is found at the end of the appropriate branch, which we call a ***leaf***, and draw as a square.

The remaining vertices are called the ***internal vertices*** of the tree. The number of comparisons done by an algorithm in a particular search is the number of internal vertices traversed in going from the top of the tree, called its ***root***, down the appropriate path to a leaf.

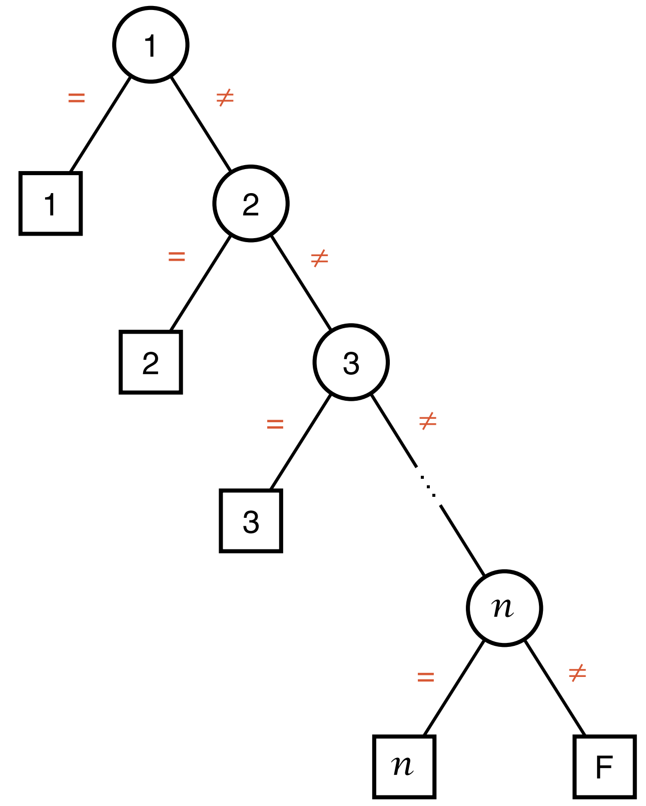
The number of branches traversed to reach a vertex from the root is called the ***level*** of the vertex. Thus the root itself has level 0, the vertices immediately below it have level 1, and so on. The largest level that occurs is called the ***height*** of the tree.

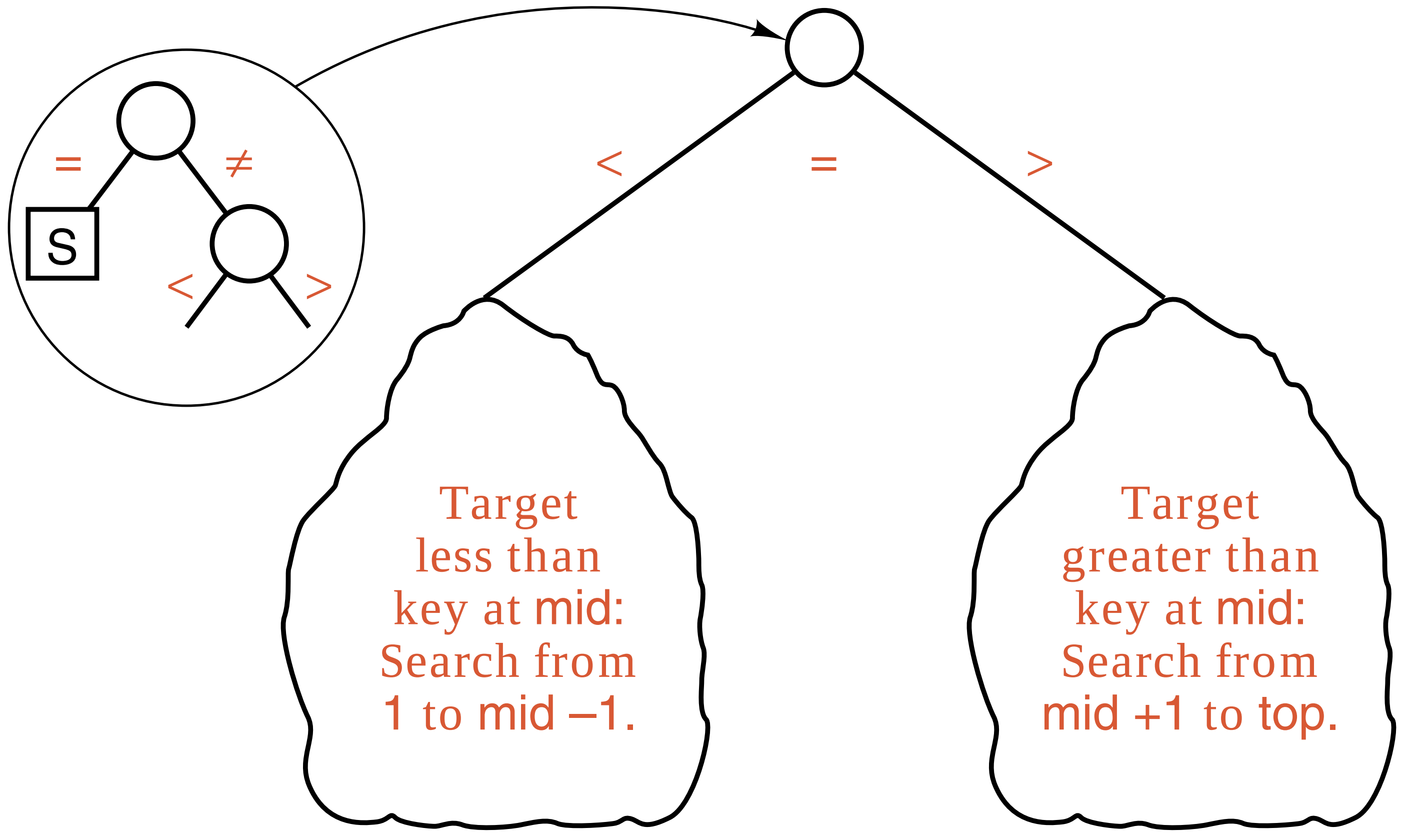
We call the vertices immediately below a vertex v the ***children*** of v and the vertex immediately above v the ***parent*** of v.

The ***external path length*** of a tree is the sum of the number of branches traversed in going from the root once to every leaf in the tree.

The ***internal path length*** is defined to be the sum, over all vertices that are not leaves, of the number of branches from the root to the vertex.

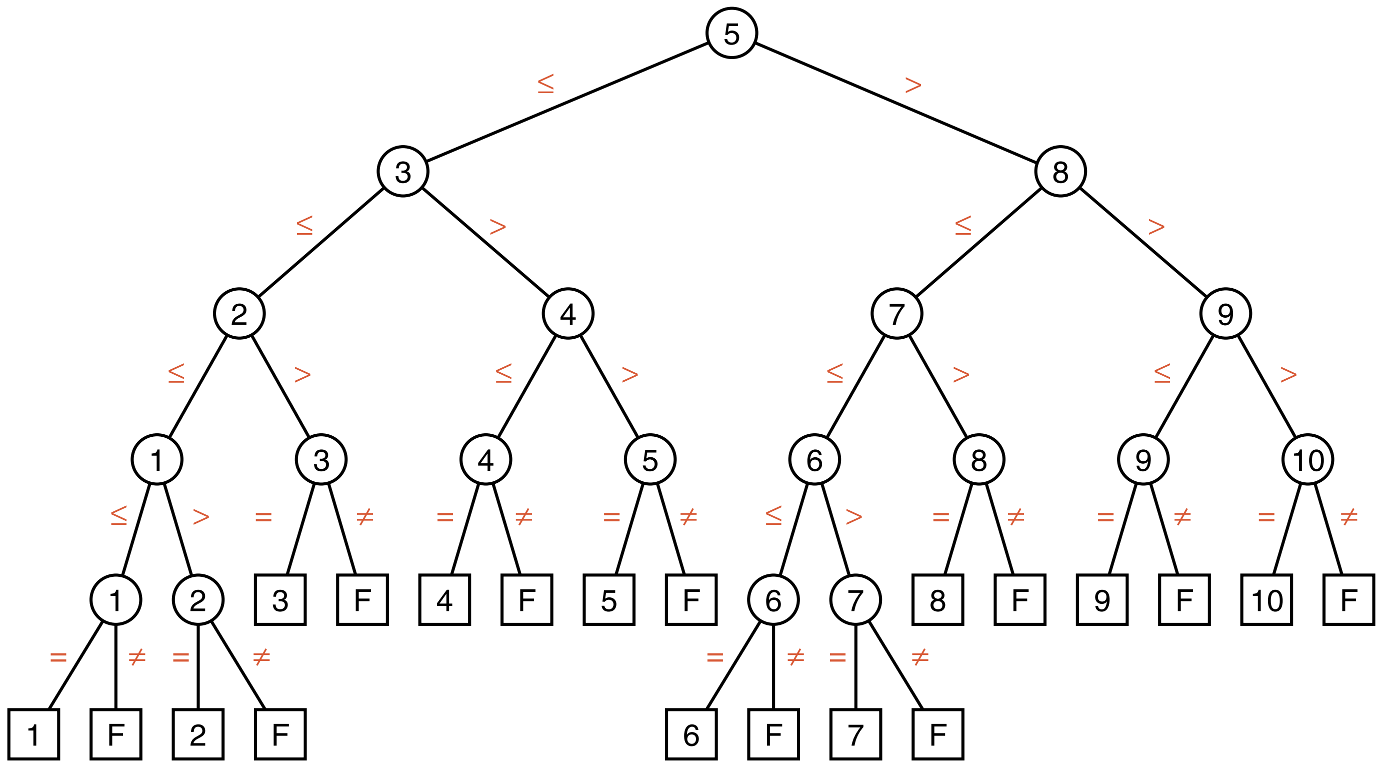
Sample Comparison Trees



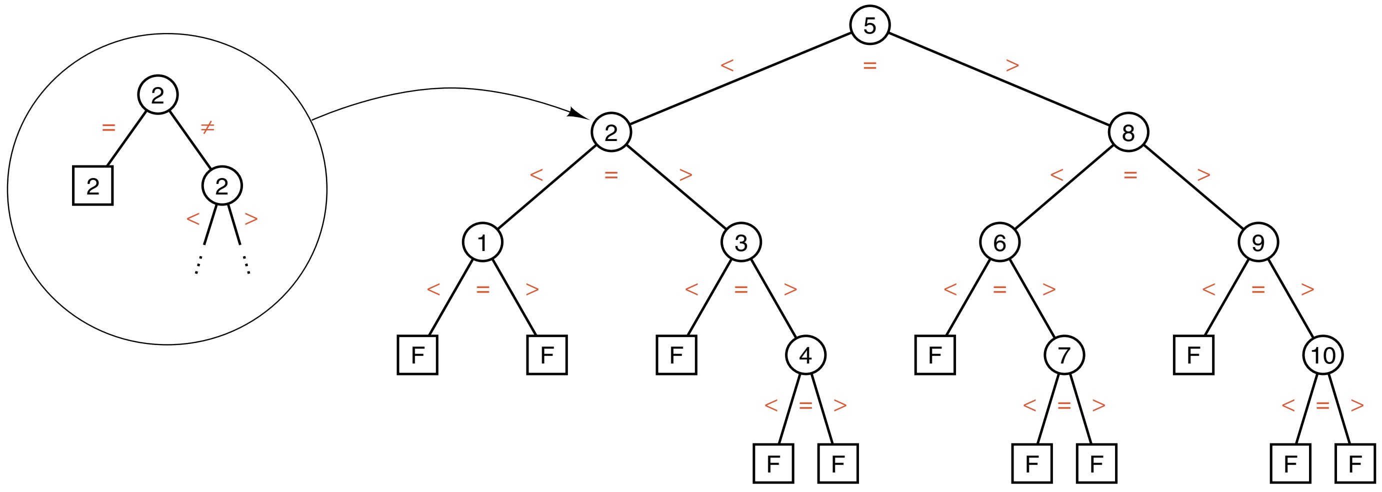

=
≠
S
<
>
<
=
>
Target less than key at mid: Search from 1 to mid −1.
Target greater than key at mid: Search from mid +1 to top.

Comparison Trees for Binary Search

First version:



Second version:



2-Trees

As ***2-tree*** is a tree in which every vertex except the leaves has exactly two children.

Lemma 7.1 The number of vertices on each level of a 2-tree is at most twice the number on the level immediately above. Hence, in a 2-tree, the number of vertices on level t is at most 2^t for $t \geq 0$.

Lemma 7.2 If a 2-tree has k vertices on level t, then $t \geq \lg k$, where lg denotes a logarithm with base 2.

Conventions

Unless stated otherwise, all logarithms will be taken with base 2.
The symbol $\lg$ denotes a logarithm with base 2,
and the symbol $\ln$ denotes a natural logarithm.
When the base for logarithms is not specified (or is not important),
then the symbol $\log$ will be used.

The ***floor*** of a real number x is the largest integer less than or equal to x, and the ***ceiling*** of x is the smallest integer greater than or equal to x. We denote the floor of x by $\lfloor x \rfloor$ and the ceiling of x by $\lceil x \rceil$.

Binary Search Analysis

The number of comparisons of keys done by binary_search_1 in searching a list of n items is approximately

$$\lg n + 1$$

in the worst case and

$$\lg n$$

in the average case. The number of comparisons is essentially independent of whether the search is successful or not.

The number of comparisons done in an unsuccessful search by binary_search_2 is approximately $2\lg(n+1)$.

In a successful search of a list of n entries, binary_search_2 does approximately

$$\frac{2(n+1)}{n}\lg(n+1) - 3$$

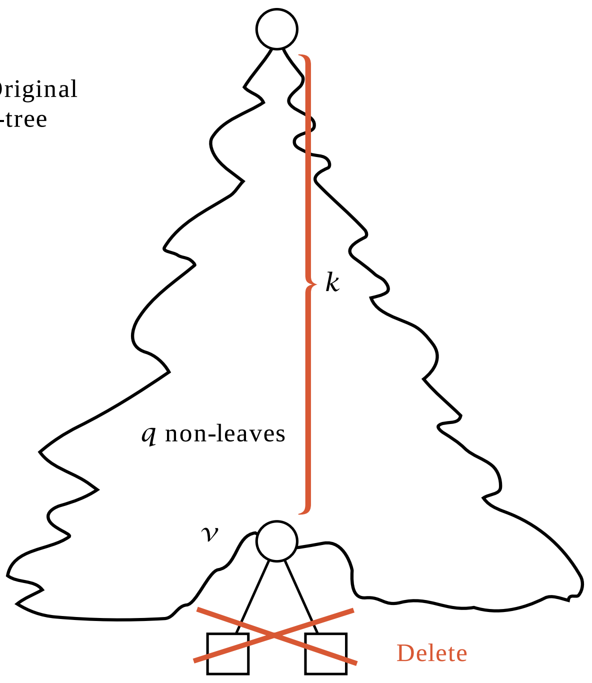
comparisons of keys.

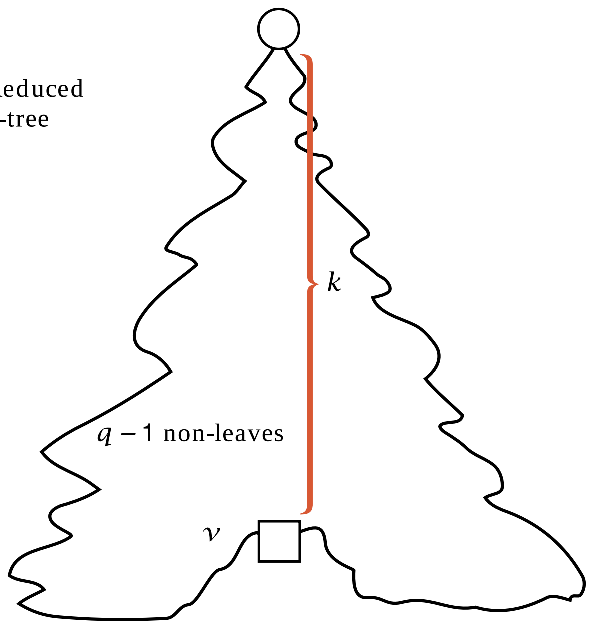
The proof of the above results requires the *path length theorem*.

	Successful search	*Unsuccessful search*
binary_search_1	$\lg n + 1$	$\lg n + 1$
binary_search_2	$2\lg n - 3$	$2\lg n$

The Path-Length Theorem

Theorem 7.3 Denote the external path length of a 2-tree by E, the internal path length by I, and let q be the number of vertices that are not leaves. Then $E = I + 2q$.

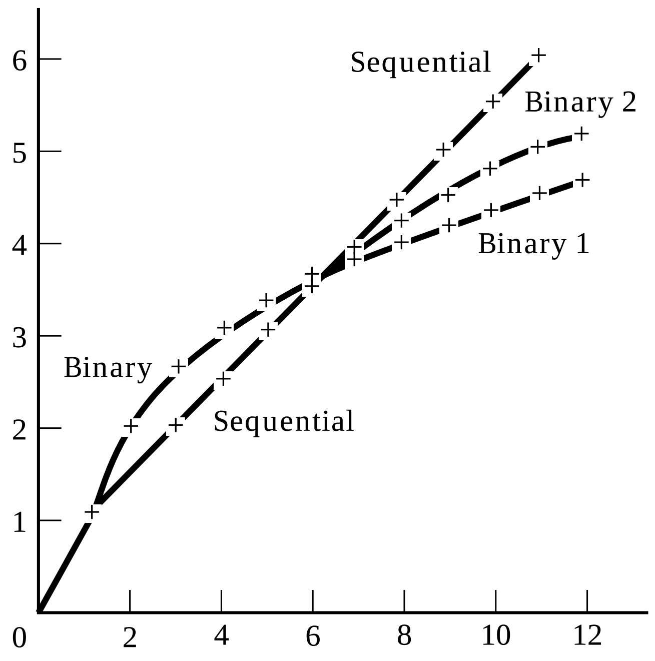




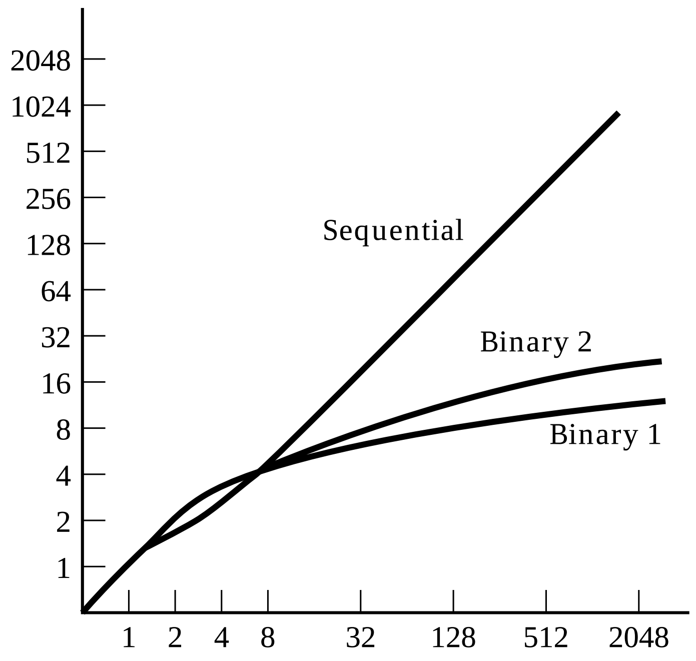
Theorem 7.4 Assume that, in searching a list of size n, the leaves of the comparison tree correspond to unsuccessful searches, the internal vertices to successful searches, and that two comparisons of keys are made for each internal vertex, except that only one is made at the vertex where the target is found. Then the average numbers S and U of key comparisons done in successful and unsuccessful searches are related by

$$S = \left(1 + \frac{1}{n}\right) U - 3.$$

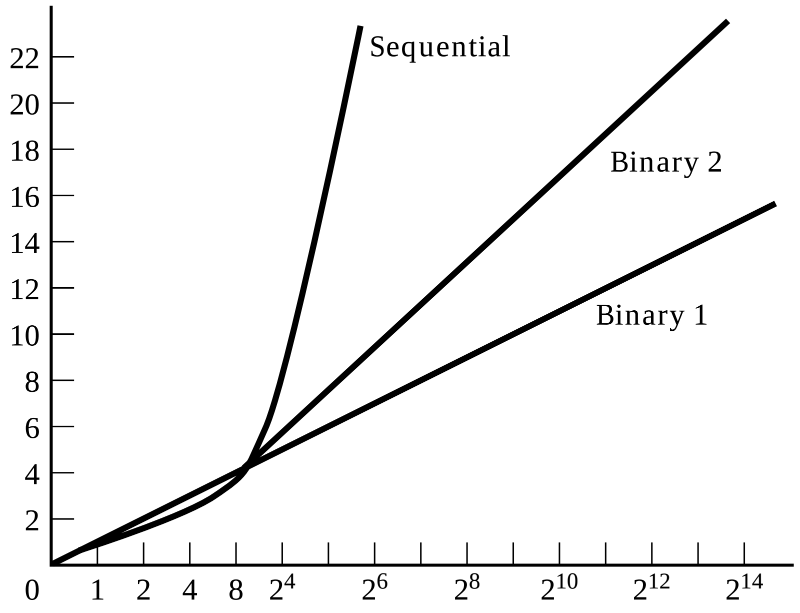
Comparison of Average Successful Searches



linear scale



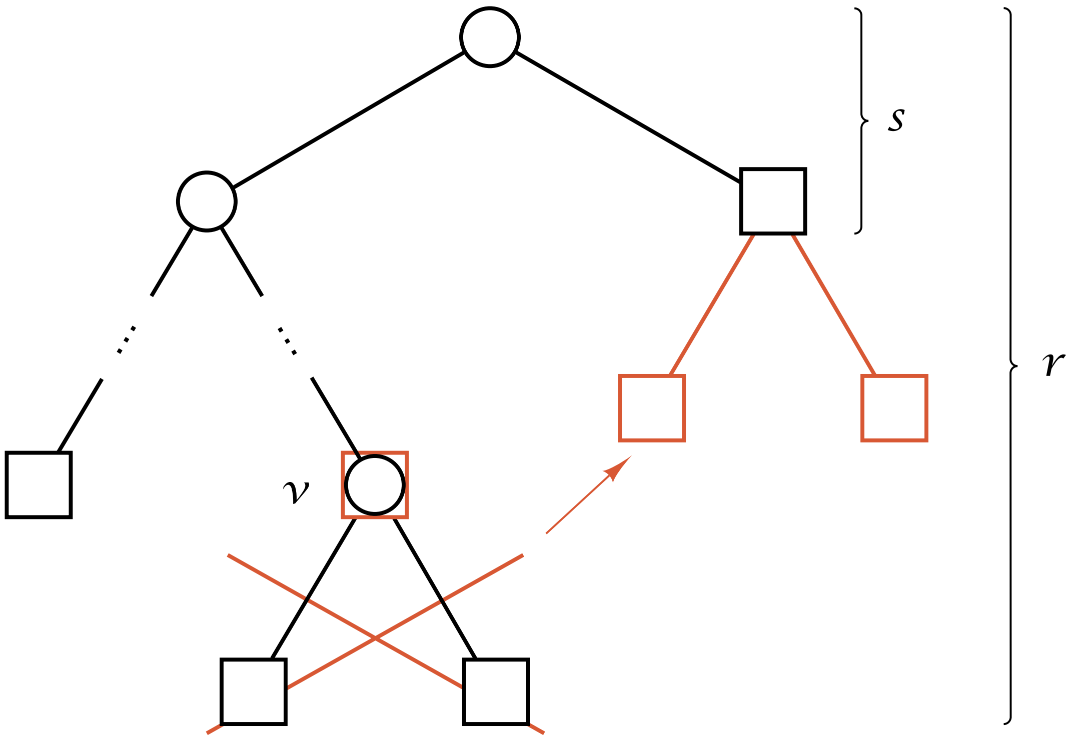
log-log scale



semilog scale

Lower Bounds on Path Length

Lemma 7.5 Let T be a 2-tree with k leaves. Then the height h of T satisfies $h \ge \lceil \lg k \rceil$ and the external path length $E(T)$ satisfies $E(T) \ge k \lg k$. The minimum values for h and $E(T)$ occur when all the leaves of T are on the same level or on two adjacent levels.



Theorem 7.6 Suppose that an algorithm uses comparisons of keys to search for a target in a list. If there are k possible outcomes, then the algorithm must make at least $\lceil \lg k \rceil$ comparisons of keys in its worst case and at least $\lg k$ in its average case.

Corollary 7.7 binary_search_1 is optimal in the class of all algorithms that search an ordered list by making comparisons of keys. In both the average and worst cases, binary_search_1 achieves the optimal bound.

Orders of Magnitude

DEFINITION ***Asymptotics*** is the study of functions of a parameter n, as n becomes larger and larger without bound.

IDEA: Suppose the function $f(n)$ measures the amount of work done by an algorithm on a problem of size n. We compare $f(n)$, for large values of n, with some well-known function $g(n)$ whose behavior we already understand. Common choices for $g(n)$:

$g(n) = 1$	Constant function
$g(n) = \log n$	Logarithmic function
$g(n) = n$	Linear function
$g(n) = n^2$	Quadratic function
$g(n) = n^3$	Cubic function
$g(n) = 2^n$	Exponential function

DEFINITION If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then:
$f(n)$ has ***strictly smaller order of magnitude*** than $g(n)$.

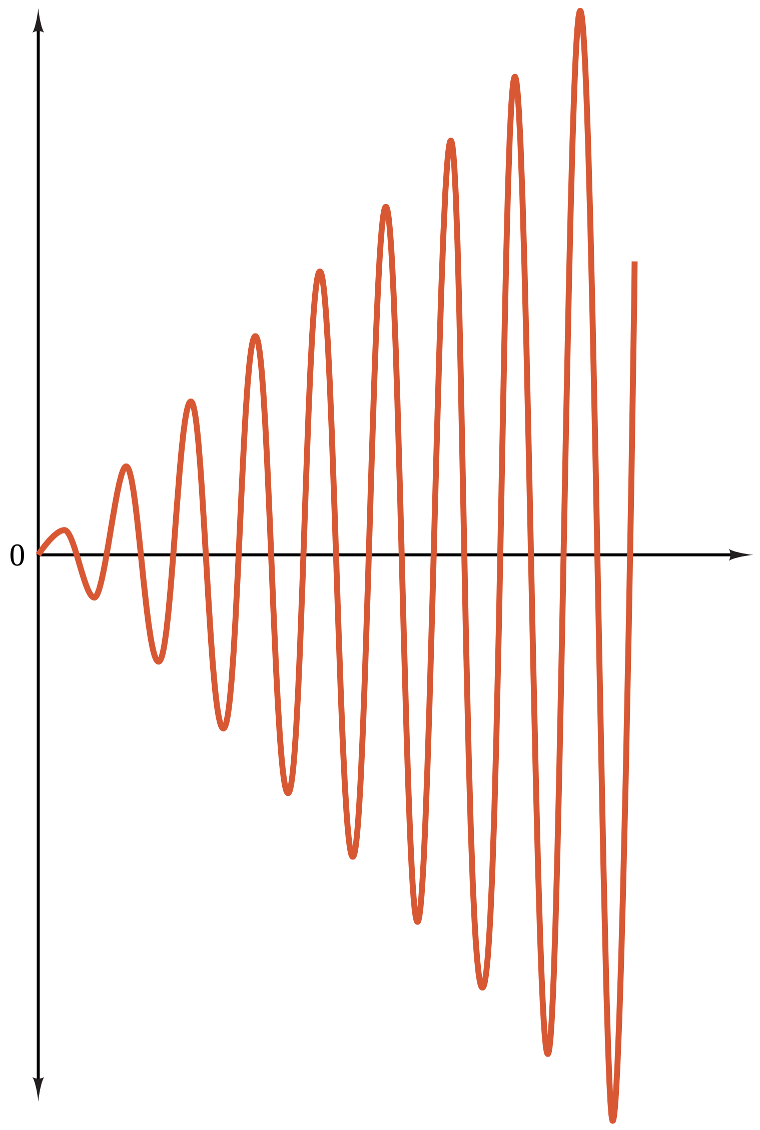
If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite and nonzero then:
$f(n)$ has ***the same order of magnitude*** as $g(n)$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then:
$f(n)$ has ***strictly greater order of magnitude*** than $g(n)$.

The second case, when $f(n)$ and $g(n)$ have the same order of magnitude, includes all values of the limit except 0 and ∞. In this way, changing the running time of an algorithm by any nonzero constant factor will not affect its order of magnitude.

Assumptions

- We assume that $f(n) > 0$ and $g(n) > 0$ for all sufficiently large n.
- We assume that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists.



We follow the convention that the limit of a function that grows larger and larger without bound *does* exist and is infinite:

Definition $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ means $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Orders of Magnitude of Common Functions

Polynomials:

If $f(n)$ is any polynomial in n with degree r, then $f(n)$ has the same order of magnitude as n^r.

If $r < s$, then n^r has strictly smaller order of magnitude than n^s.

Logarithms:

The order of magnitude of a logarithm does not depend on the base for the logarithms.

Since the base for logarithms makes no difference to the order of magnitude, we shall generally write log rather than lg or ln in any order-of-magnitude expression.

$\log n$ has strictly smaller order of magnitude than any positive power n^r of n, $r > 0$.

This result can be proved by using *L'Hôpital's Rule*.

L'Hôpital's Rule

Theorem 7.8 **L'Hôpital's Rule** Suppose that:

- $f(x)$ and $g(x)$ are differentiable functions for all sufficiently large x, with derivatives $f'(x)$ and $g'(x)$, respectively.
- $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to\infty} g(x) = \infty$.
- $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists.

Then $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ exists and $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$.

Exponential functions:

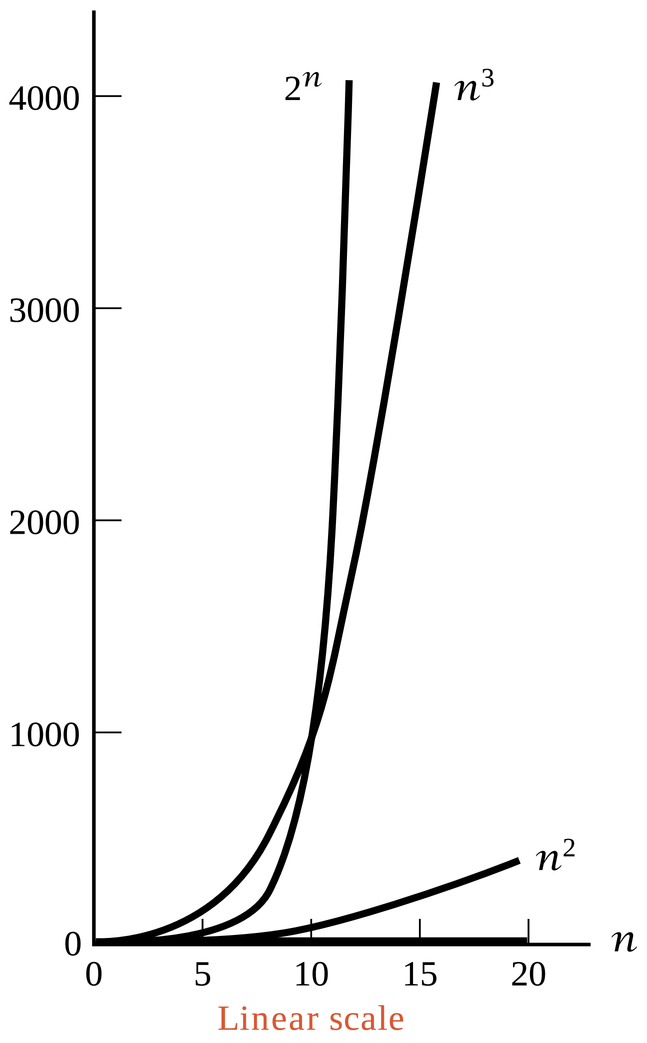
Any exponential function a^n for any real number $a > 1$ has strictly greater order of magnitude than any power n^r of n, for any positive integer r.

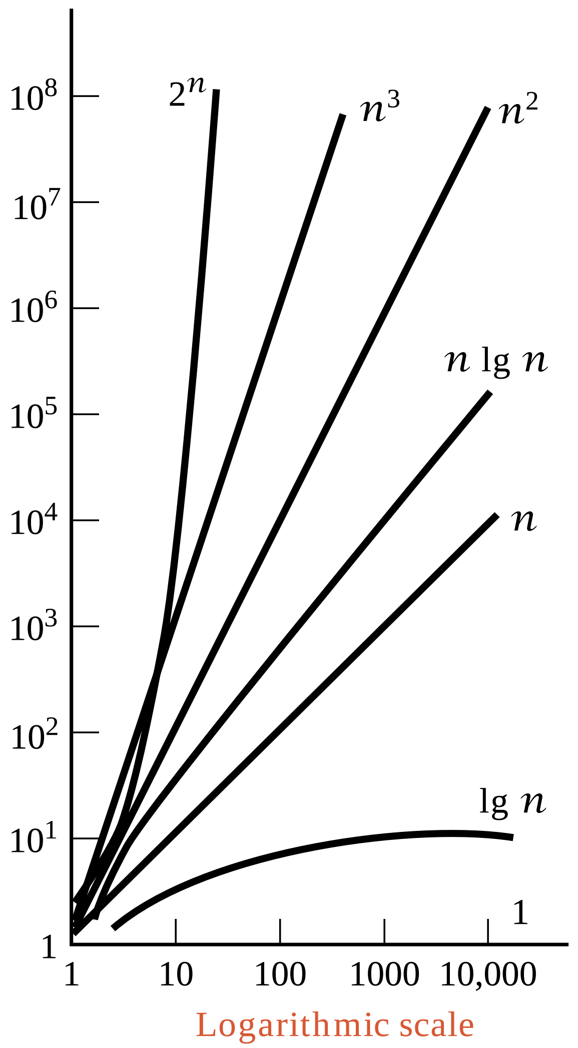
If $0 \le a < b$ then a^n has strictly smaller order of magnitude than b^n.

The function $n \log n$:

The function $n \log n$ has strictly greater order of magnitude than n but strictly smaller order of magnitude than any power n^r for any $r > 1$.

Growth Rates of Common Orders





n	1	$\lg n$	n	$n \lg n$	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1000	1024
100	1	6.64	100	664	10,000	1,000,000	1.27×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.07×10^{301}

The Big-O and Related Notations

Notation: $f(n)$ *is*	*Meaning:* *Order of* f *compared to* g *is*		*Value of* $\lim_{n\to\infty}\big(f(n)/g(n)\big)$
$o\big(g(n)\big)$	$<$	*strictly smaller*	0
$O\big(g(n)\big)$	$\leq$	*smaller or equal*	*finite*
$\Theta\big(g(n)\big)$	$=$	*equal*	*nonzero finite*
$\Omega\big(g(n)\big)$	$\geq$	*larger or equal*	*nonzero*

These notations are pronounced "little oh," "Big Oh," "Big Theta," and "Big Omega," respectively.

Examples:

- On a list of length n, sequential search has running time $\Theta(n)$.
- On an ordered list of length n, binary search has running time $\Theta(\log n)$.
- Retrieval from a contiguous list of length n has running time $O(1)$.
- Retrieval from a linked list of length n has running time $O(n)$.
- Any algorithm that uses comparisons of keys to search a list of length n must make $\Omega(\log n)$ comparisons of keys (Theorem 7.6).
- Any algorithm for the Towers of Hanoi requires time $\Omega(2^n)$ in order to move n disks.

Algorithm Analyses

- If $f(n)$ is a polynomial in n of degree r, then $f(n)$ is $\Theta(n^r)$.
- If $r < s$, then n^r is $o(n^s)$.
- If $a > 1$ and $b > 1$ then $\log_a(n)$ is $\Theta\big(\log_b(n)\big)$.
- $\log n$ is $o(n^r)$ for any $r > 0$.
- For any real number $a > 1$ and any positive integer r, n^r is $o(a^n)$.
- If $0 \le a < b$ then a^n is $o(b^n)$.

Keeping the dominant term:

We define $f(n) = g(n) + O\big(h(n)\big)$ to mean that $f(n) - g(n)$ is $O\big(h(n)\big)$.

More precise form:

- For a successful search in a list of length n, sequential search has running time $\frac{1}{2}n + O(1)$.
- For a successful search in an ordered list of length n, binary search has running time $2\lg n + O(1)$.
- Retrieval from a contiguous list of length n has running time $O(1)$.
- Retrieval from a simply linked list of length n has average running time $\frac{1}{2}n + O(1)$.

Pointers and Pitfalls

1. In designing algorithms be very careful of the extreme cases, such as empty lists, lists with only one item, or full lists (in the contiguous case).
2. Be sure that all your variables are properly initialized.
3. Double check the termination conditions for your loops, and make sure that progress toward termination always occurs.
4. In case of difficulty, formulate statements that will be correct both before and after each iteration of a loop, and verify that they hold.
5. Avoid sophistication for sophistication's sake. Whenever a simple method is adequate for your application, use it.
6. Don't reinvent the wheel. If a ready-made function is adequate for your application, use it.
7. Sequential search is slow but robust. Use it for short lists or if there is any doubt that the keys in the list are properly ordered.
8. Be extremely careful if you must reprogram binary search. Verify that your algorithm is correct and test it on all the extreme cases.
9. Drawing trees is an excellent way both to trace the action of an algorithm and to analyze its behavior.
10. Rely on the big-O analysis of algorithms for large applications but not for small applications.