Imperative Programming

Week 6

Complexity of algorithms

Complexity of an algorithm

=

The number of ‘basic’ computational steps that an algorithm performs to compute its output given the input.

Complexity theory studies the relation between the ‘size’ of the input and the number of basic computation steps to compute the output.

How to determine complexity?

It is clear that computing

`234456597356 * 976895793565`

is more work than computing

`15 * 10`

- Still, *the general method* (the algorithm) is the same.

scalability

Execution time scales with the speed of the computer:

- If computer B is a factor k faster than computer A, then the expected execution time for a given problem on computer B is a factor k smaller than on computer A.

This scalability property is, however, in general not true for the size of the input of an algorithm.

Example: In general, it is not true that sorting an array that is a factor k longer than some other array takes a factor k longer in execution time!

The scalability depends on the algorithm that you use!

Generating solutions

Given natural numbers **a**, **b**, **c**, **N**, and **M**, such that **a != c** or **b != c**.

Generate all solutions of the equation:

a*x + b*y + c*z == N, where **x + y + z == M**.

Generating solutions

Solution 1: try all combinations.

```
void solution1(int M, int N, int a, int b, int c) {
   int x, y, z;
   for (x=0; x <= M; x++) {      /* M+1 iterations */
     for (y=0; y <= M; y++) {    /* M+1 iterations */
       for (z=0; z <= M; z++) { /* M+1 iterations */
         if ((x + y + z == M) && (a*x + b*y + c*z == N)) {
           printf("x=%d, y=%d, z=%d\n", x, y, z);
         }
       }
     }
   }
 }
```

Generating solutions

The complexity of nested for-loops:

The body of the *outer* loop is executed M+1 times, because **x** loops through all values from 0 to **M** (including **M**).

In the body of the outer loop, there is a second for-loop. For each value of **x**, the body of the second loop is executed **M+1** times as well.

So, the body of the second loop is executed **(M+1)*(M+1)** times in total.

The body of the second loop, however, contains a third loop, that for each combination of **x** and **y** is executed **M+1** times.

Generating solutions

We conclude that the body of the innermost loop is executed
$(M+1)*(M+1)*(M+1)=(M+1)^3$ times.

In the body there are *4 additions*, *3 multiplications*, *2 comparisons* and a *boolean-operator* ***&&***.

So, we find a total of $10*(M+1)^3$ basic operations.

M	1	2	3	10	20	30	99	999
10(M+1)^3	80	270	640	13310	92610	297910	10^7	10^10

Generating solutions

Solution 2: a small improvement

```
void solution2(int M, int N, int a, int b, int c) {
   int x, y, z;
   for (x=0; x <= M; x++) {      /* M+1 iterations */
     for (y=0; y <= M-x; y++) {    /* M+1-x iterations */
       for (z=0; z <= M-x-y; z++) { /* M+1-x-y iterations */
         if ((x + y + z == M) && (a*x + b*y + c*z == N)) {
           printf("x=%d, y=%d, z=%d\n", x, y, z);
         }
       }
     }
   }
 }
```

Four useful lemmas

$$\sum_{i=0}^{n} 1 = n+1$$

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

$$\sum_{i=0}^{n} i^2 = \sum_{i=0}^{n} (n-i)^2 = n(n+1)(2n+1)/6$$

$$\sum_{i=0}^{n} i^3 = \sum_{i=0}^{n} (n-i)^3 = (n(n+1)/2)^2$$

Generating solutions

We consider the inner loop:

```
for (z=0; z<=M-x-y; z++)
```

For a pair x,y this loop is executed M+1-x-y times.

We introduce: $S(x,y) = M+1-x-y$

So, we can regard the algorithm as:

```
for (x=0; x <= M; x++) {
    for (y=0; y <= M-x; y++) {
        /* Perform S(x,y) steps */
    }
}
```

How many computational steps does this take?

$$\sum_{x=0}^{M}\sum_{y=0}^{M-x} S(x,y) = \sum_{x=0}^{M}\sum_{y=0}^{M-x}(M+1-x-y)$$

Generating solutions

$$\sum_{x=0}^{M}\sum_{y=0}^{M-x}(M+1-x-y)=\sum_{x=0}^{M}\sum_{y=0}^{M-x}((M+1-x)-y)=\sum_{x=0}^{M}\left(\sum_{y=0}^{M-x}(M+1-x)-\sum_{y=0}^{M-x}y\right)=$$

$$\sum_{x=0}^{M}\left((M+1-x)\sum_{y=0}^{M-x}1-\sum_{y=0}^{M-x}y\right)=\sum_{x=0}^{M}\left((M+1-x)(M+1-x)-\frac{(M-x)(M+1-x)}{2}\right)=$$

$$\sum_{x=0}^{M}\frac{2(M+1-x)^2-(M-x)(M+1-x)}{2}=\frac{1}{2}\sum_{x=0}^{M}\left(2(M+1-x)^2-((M+1-x)-1)(M+1-x)\right)=$$

$$\frac{1}{2}\sum_{x=0}^{M}\left((M+1-x)^2+(M+1-x)\right)=\frac{1}{2}\sum_{x=0}^{M}\left((M+1)^2-2(M+1)x+x^2+(M+1-x)\right)=$$

$$\frac{1}{2}\sum_{x=0}^{M}\left((x^2-(2M+3)x+M^2+3M+2)\right)=\frac{1}{2}\left(\sum_{x=0}^{M}x^2-(2M+3)\sum_{x=0}^{M}x+(M^2+3M+2)\sum_{x=0}^{M}1\right)=$$

$$\frac{1}{2}\left(\frac{M(M+1)(2M+1)}{6}-\frac{(2M+3)M(M+1)}{2}+(M^2+3M+2)(M+1)\right)=$$

$$\frac{1}{12}\left((M^2+M)(2M+1)-3(2M+3)(M^2+M)+6(M^2+3M+2)(M+1)\right)=$$

$$\frac{1}{12}\left((2M^3+3M^2+M)-3(2M^3+5M^2+3M)+6(M^3+4M^2+5M+2)\right)=$$

$$\frac{1}{12}\left(2M^3+12M^2+25M+12\right)$$

Generating solutions

So, the number of steps is: $\frac{1}{12}\left(2M^3 + 12M^2 + 25M + 12\right)$

Hence, the number of operations is: $\frac{10}{12}\left(2M^3 + 12M^2 + 25M + 12\right)$

This is better than solution: $10(M+1)^3$

Generating solutions

Solution 3: A much better algorithm.

```
void solution3(int M, int N, int a, int b, int c) {
   int x, y, z;
   for (x=0; x <= M; x++) {      /* M+1 iterations */
     for (y=0; y <= M-x; y++) {  /* M+1-x iterations */
       z = M-x-y; /* now we are sure that: x+y+z==M */
       if (a*x + b*y + c*z == N) {
         printf("x=%d, y=%d, z=%d\n", x, y, z);
       }
     }
   }
 }
```

Generating solutions

Again, we consider the inner loop:

```
for (y=0; y <= M-x; y++)
```

For each **x** this loop is executed **M+1-x** times.

We introduce : $T(x) = M + 1 - x$

So, we can regard the algorithm as:

```
for (x=0; x <= M; x++) {
   /* Perform T(x) steps */
}
```

How many computational steps does this take?

$$\sum_{x=0}^{M} T(x) = \sum_{x=0}^{M} (M+1-x) = \sum_{x=0}^{M} (M+1) - \sum_{x=0}^{M} x = (M+1)^2 - \frac{M(M+1)}{2} =$$

$$M(M+1) + (M+1) - \frac{M(M+1)}{2} = \frac{M(M+1) + 2(M+1)}{2} = \frac{M^2 + 3M + 2}{2}$$

A very efficient solution

So far, we did not use: a!=c or b!=c.

We search for a*x + b*y + c*z == N, with x + y + z == M.

Substitute z == M - x- y in a*x + b*y + c*z == N and we find:
a*x + b*y + c*(M-x-y) == N

Some calculus yields: (a-c)*x + (b-c)*y == N-c*M

So, given x we find y == (N+ (c-a)*x -c*M)/(b-c)
(assuming b != c)

Analogous, given y we find x == (N+ (c-b)*y -c*M)/(a-c)
(assuming a != c)

A very efficient solution

```
void solution4(int M, int N, int a, int b, int c) {
  int x, y, z;
  if (b != c) {
    for (x=0; x <= M; x++) {  /* M+1 iterations */
      y = (N + (c-a)*x - c*M)/(b-c);
      z = M - x - y;
      if (a*x + b*y + c*z == N) {
        printf("x=%d, y=%d, z=%d\n", x, y, z);
      }
    }
  } else {
    /* a != c */
    for (y=0; y <= M; y++) {  /* M+1 iterations */
      x = (N + (c-b)*y - c*M)/(a-c);
      z = M - x - y;
      if (a*x + b*y + c*z == N) {
        printf("x=%d, y=%d, z=%d\n", x, y, z);
      }
    }
  }
}
```

Comparing the algorithms

Solution 1: $10(M+1)^3$

Solution 2: $\frac{10}{12}\left(8M^3+12M^2+25M+12\right)$

Solution 3: $9\frac{M^2+3M+2}{2}$ (9 operations in inner loop)

Solution 4: $17\frac{M+1}{2}$ (17 operations in inner loop)

The fourth solution is clearly the best. For large values of M, the solutions 1 and 2 are comparable (the factor becomes irrelevant). Solution 3 is, however, about M times faster than the solutions 1 and 2, while solution 4 is even about M*M times faster!

Order of complexity

- We can compute the number of computational steps as a function of the input size in great detail. This leads for any algorithm A to an expression C(A) that yields the complexity of A.
- Using such an expression we can compare the time complexity of algorithms:
  - A is better than B if C(A)<C(B).
- If we are not interested in the complexity in great detail, but we do wish to say something about complexities, then we resort to *order calculations*.
- Order-calculations are inexact. But, they do have the advantage that you know the scalability of an algorithm up to some factor. This way we can introduce a hierarchy of *algorithm classes.*

Order of complexity

- In *order calculations*, we only look at the part of the expressions C(A) that dominates for (very) large input.

- For example, let

$$C(A) = 3N^3 + 4N^2 + 5N + 12$$

then, for large N, this expressions is dominated by the first term. We say that the algorithm has a *cubic* time complexity, or ‘A is in $O(N^3)$’, or ‘A is of the order N^3’.

Big O-notation: Bounded above

- We say that some algorithm A is *bounded above* if its (exact) time complexity is at most some expression g(N).

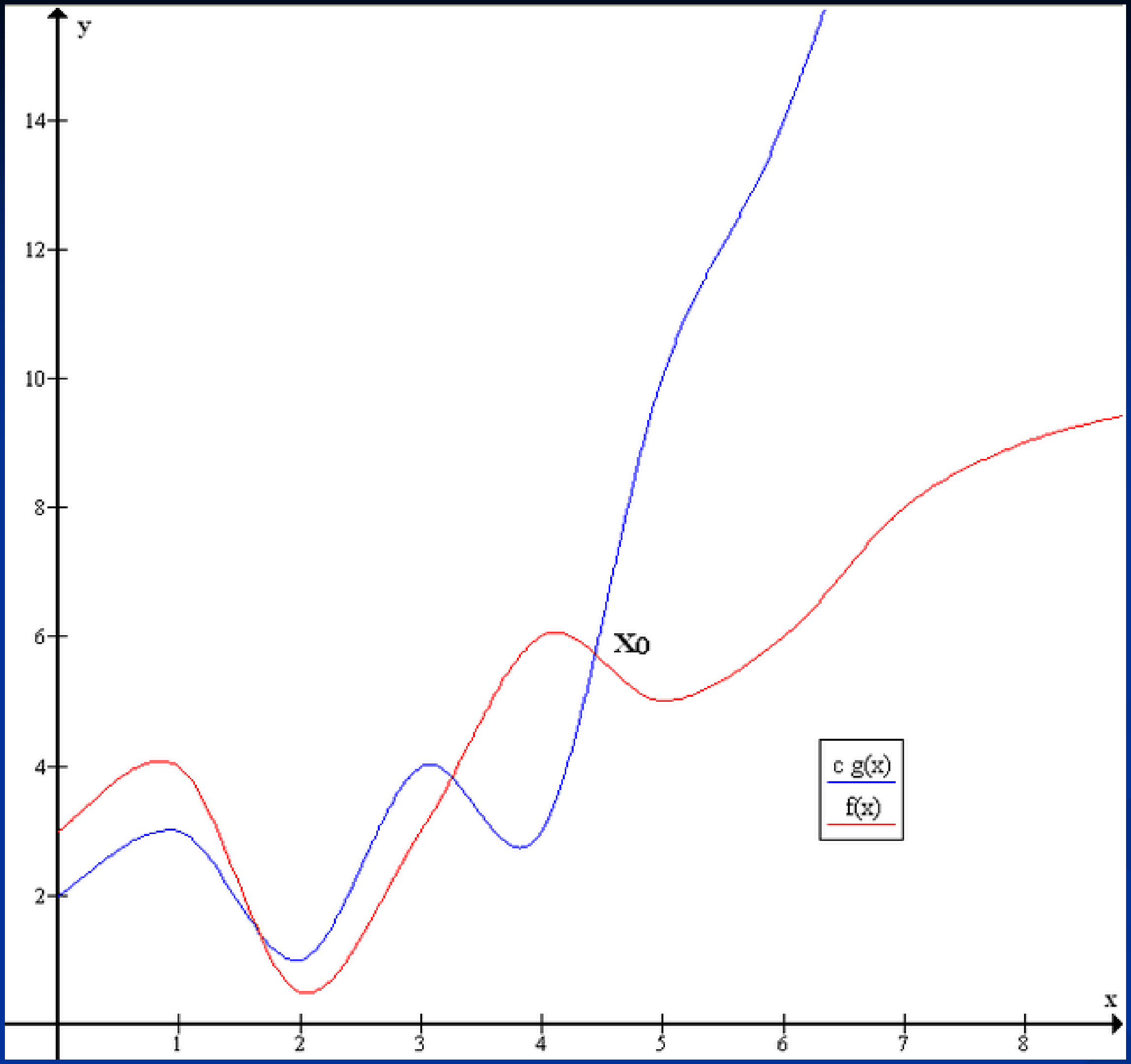
- This is denoted with the so-called "big-O" notation (but it is actually the Greek capital letter omicron).

- We write $A \in O(g(N))$, which means that the exact number of computation steps that A performs *is at most* c*g(N)*.*

- *This expression is a function of N: the input size*

Formally:

$$A \in \mathrm{O}(g(n)) \Leftrightarrow \exists_{c>0} \exists_{N>0} \forall_{n \geq N}\, 0 \leq A(n) \leq c \cdot g(n)$$



Example of Big O notation: $f(x) \in O(g(x))$ as there exists $c > 0$ (e.g. $c = 1$) and N (e.g., $N = 5$) such that $f(x) < c*g(x)$ whenever $x > N$.

Why compute big O?

- The big O notation tells us what we may expect for the scalability of the runtime if we scale the input: for sufficiently large inputs the big O term dominates all other terms in C(A).

- It is much easier to determine orders than exact expressions for C(A).

- It is much easier to compare the quality of algorithms. Algorithms from the same big-O class, will scale approximately the same (up to some constant factor).

Other bounds

- Bounded below:

$$A \in \Omega(g(n)) \Leftrightarrow \exists_{c>0} \exists_{N>0} \forall_{n \geq N}\, 0 \leq c \cdot g(n) \leq A(n)$$

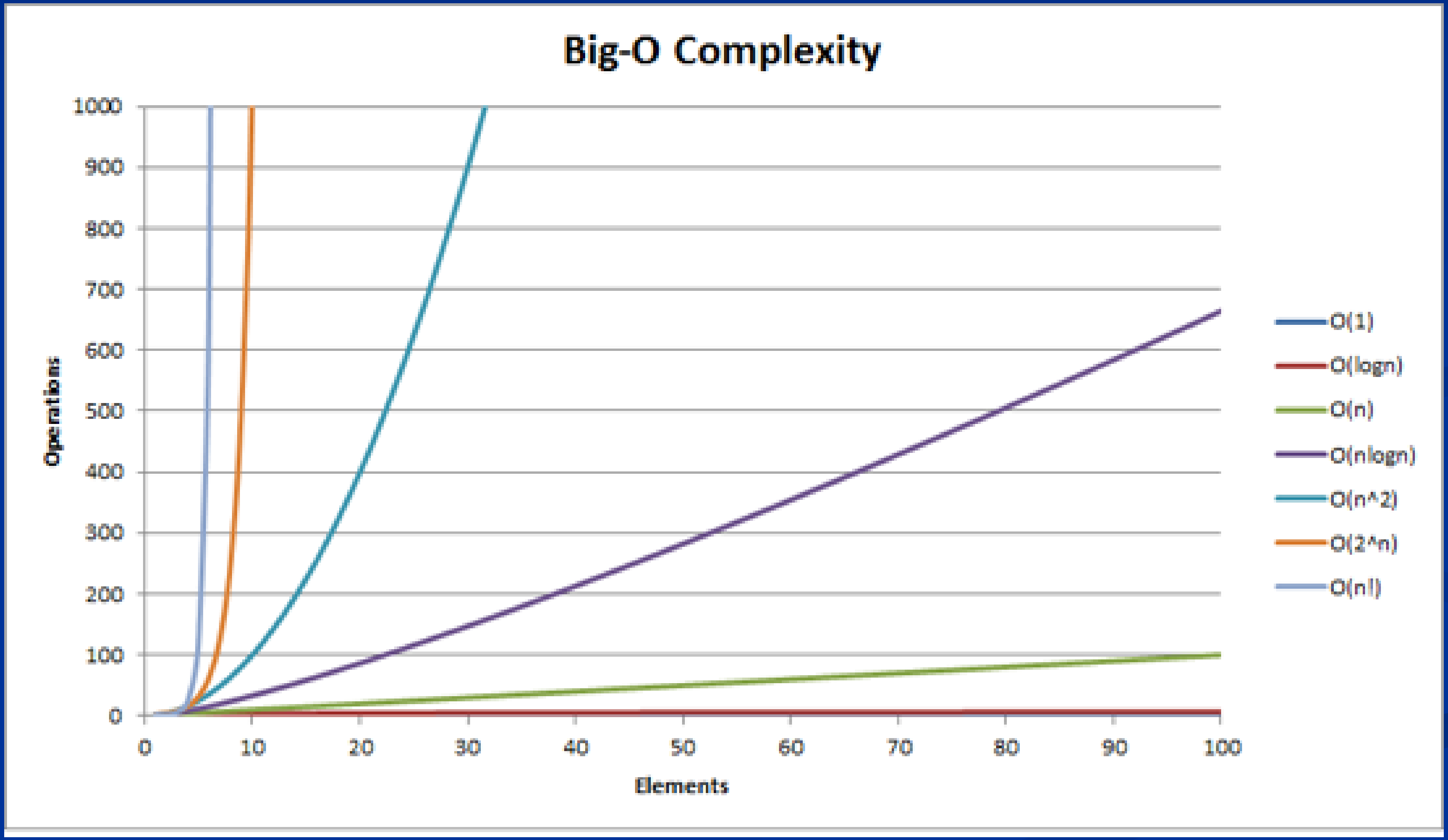
- Bounded in between (most exact):

$$A \in \Theta(g(n)) \Leftrightarrow \exists_{c,d>0} \exists_{N>0} \forall_{n \geq N}\, 0 \leq c \cdot g(n) \leq A(n) \leq d \cdot g(n)$$

Common complexities

- $O(1)$: the computation time is independent of the size of the input ("constant time complexity").
- $O(n)$: computation time scales linearly with the size of the input ("linear time complexity").
- $O(n^2)$: computation time scales quadratic with the size of the input ("quadratic time complexity").
- $O(\log n)$: computation time scales with the logarithm of the size of the input , i.e. if n doubles then the computation time scales with a constant factor ("logarithmic time complexity").
- $O(n \log n)$: This is the complexity of several advanced sorting algorithms.
- $O(2^n)$: execution time increases exponentially with the size of the input ("exponential time complexity").

Common complexities



Common complexities

- O(1): constant time complexity

- Example: Chess board distance between two grid coordinates:

```
int chessboardDistance(int x0, int y0, int x1, int y1) {
  int dx = (x1 > x0 ? x1 - x0 : x0 - x1);
  int dy = (y1 > y0 ? y1 - y0 : y0 - y1);
  return (dx > dy ? dx : dy);
}
```

Linear search

We search in an array a[ ] the smallest index i for which a[i] == value. If such an i does not exist, we return -1.

```
int linearSearch(int length, int a[], int value) {
  int i;
  for (i=0; i < length; i++) {
    if (a[i] == value) {
      break;
    }
  }
  return (i == length ? -1 : i);
}
```

This algorithm takes in the worst-case length steps, i.e. linear search has a linear time complexity (hence, its name).

Common complexities

- O(n): linear time complexity

```
int power(int g, int n) {
    int i, res = 1;
    for (i=0; i < n; i++){
      res = g*res;
    }
    return res;
}
```

Selection Sort: $O(n^2)$

- $O(n^2)$: quadratic time complexity

```
void swapElements(int i, int j, int a[]) {
  int h = a[i];
  a[i] = a[j];
  a[j] = h;
}

void selectionSort(int length, int a[]) {
  int i, j, smallest;
  for (i=0; i < length; i++) {
    /* determine index of minimum in interval [i,length) */
    smallest = i;
    for (j=i+1; j < length; j++) {
      if (a[j] < a[smallest]) {
        smallest = j;
      }
    }
    swapElements(i, smallest, a);
  }
}
```

Common complexities

- $O(2^n)$: exponential time complexity

```
int fib(int n) {
  /* returns fibonacci(n) */
  if (n < 2) {
    return n;
  }
  return fib(n-2) + fib(n-1);
}
```

For the number of computation steps we find:

•$S(0) = S(1) = 1;$

•$S(n) = S(n-2) + S(n-1) \leq 2S(n-1)$

Prove yourself (using induction) that: $2^{n/2} \leq S(n) \leq 2^n$

Majority vote

Write a function that, given an array parameter `int arr[]`, computes whether there is a value in `arr` that has the majority, i.e. the number of times that it occurs is more than half of the length of the array.

```
int hasMajority (int length, int arr[]) {
  int i, j, counter;
  for (i=0; i < length; i++) {
    counter = 0;
    for (j=0; j < length; j++) {
      if (arr[j] == arr[i]) {
        counter++;
      }
    }
    if (2*counter > length) {
      return 1;  /* TRUE */
    }
  }
  return 0;  /* FALSE */
}
```

Majority Vote

This algorithm uses order $length^2$ comparisons.

We can easily make this algorithm twice as fast, by starting the inner loop from **i** (instead of **0**).
But this is only a (constant) factor of **2**!

So, both versions of the algorithm have a quadratic time complexity.

Efficient Majority Vote

We make the following observation: if **x** has a majority, then we can reduce the size of the array by two elements: we cancel an occurrence of x against a non-occurrence of x. In this new array, x has still the majority.

[1 3 2 1 5 1 1] ⇨ [2 1 5 1 1] ⇨ [5 1 1] ⇨ [1]

[1 3 2 1 5 1 2] ⇨ [2 1 5 1 2] ⇨ [5 1 2] ⇨ [2]

We will not really reduce the size of the array, and use a counter **`surplus`** instead.

Majority Vote

```
int hasMajority (int length, int arr[]) {
   int candidate, counter, surplus = 0;
   int i;
   for (i=0; i < length; i++) {
     if (surplus == 0) { /* new candidate */
       candidate = arr[i];
       surplus  = 1;
     } else { /* we have a candidate already */
       if (arr[i] == candidate) { /* another vote */
         surplus++;
       } else { /* cancel out votes */
         surplus--;
       }
     }
   }
   /* if there is a majority, then we know the candidate */
   counter = 0;
   for (i=0; i < length; i++) {
     if (a[i] == candidate) {
       counter++;
     }
   }
   /* does candidate have a majority? */
   return (2*counter > length);
}
```

Majority Vote

Note that we make two passes through the array. The total number of inspections is therefore **2*length**.

So, we reduce the complexity from quadratic to linear: a big improvement.

Common complexities

- O(log n): logarithmic time complexity

```
int power(int g, int n) {
    int x = 1;
    while (n != 0) {
        if (n%2 == 1){
            x = g*x;
        }
        g = g*g;
        n = n / 2;
    }
    return x;
}
```

```
int power(int g, int m) {
  if (m==0) {
    return 1;
  }
  if (m%2 == 0) {
    return power(g*g, m/2);
  }
  return g*power(g, m-1);
}
```

Binary search

We can search in an array much faster if it is sorted (think of a dictionary, a phone book or an index in a book).

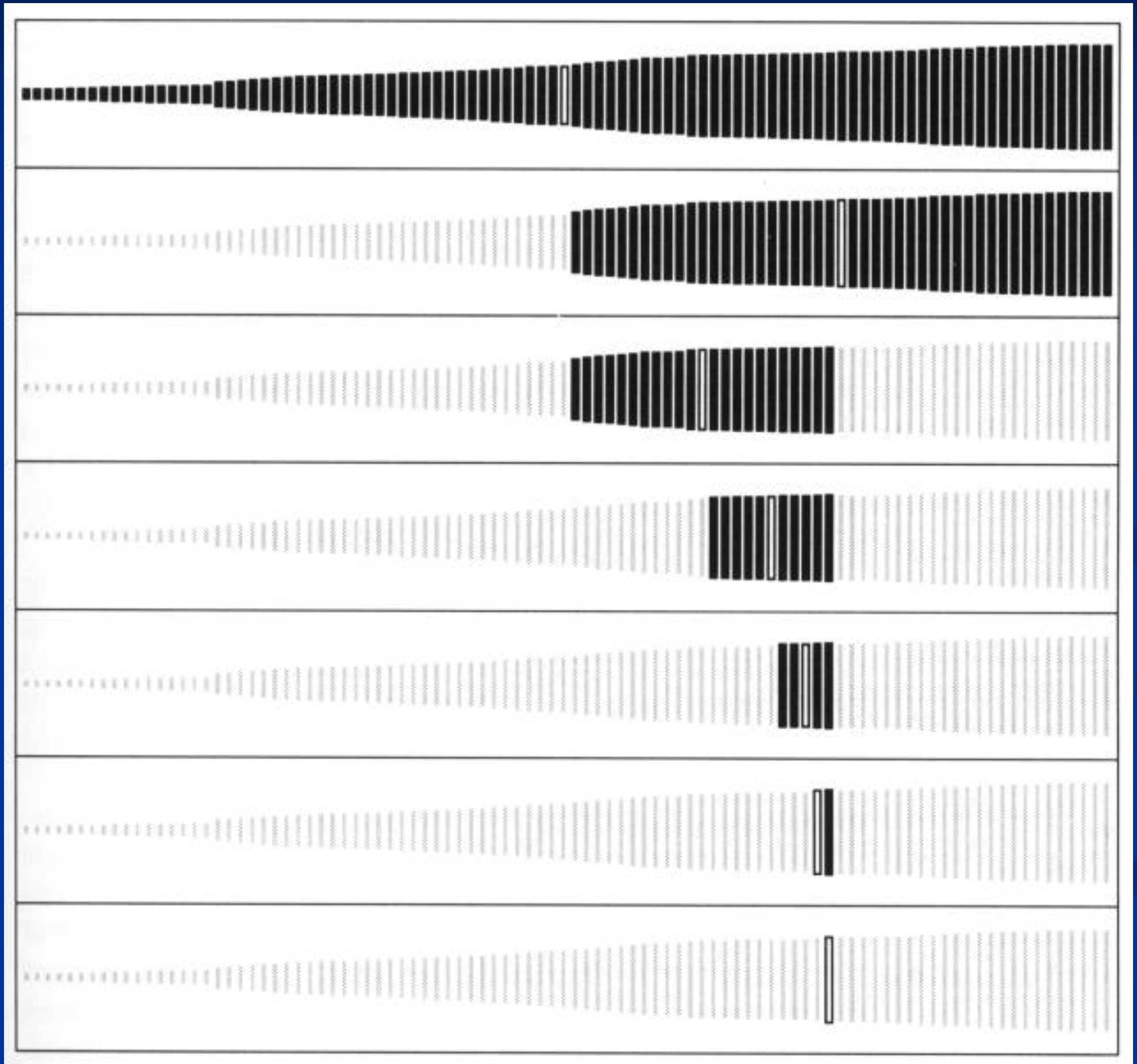
We assume:

`a[0] <= a[1] <= … <= a[length-1].`

We introduce two variables **left** and **right** to maintain the half-open search interval [left,right).

We make sure that the element that we search for cannot occur outside this interval.

Binary search



Binary search

We choose mid in the middle of the interval [left, right).

•If value < a[mid], then value must be searched for to the left of mid, i.e. we can replace right by mid.

•If a[mid]<=value, then value must be searched for to the right of mid-1, i.e. we can replace left by mid.

This process stops if there remains only one element in the search interval, i.e. when left==right–1. The only thing left is to check whether that element equals value or not.

Recursive Binary search

```
int recBinarySearch(int left, int right, int a[], int value) {
  int mid;
  /* 0 <= left < right */

  if (left == right - 1) { /* base case */
    return (a[left] == value ? left : -1);
  }

  /* 0 <= left+1 < right */
  mid = (left + right)/2;
  /* right-left > 1 implies left < mid < right */
  if (value < a[mid]){
    right = mid;
  } else {
    left = mid;
  }
  return recBinarySearch(left, right, a, value);
}

int binarySearch(int length, int a[], int value) {
  return (length == 0 ? -1 : recBinarySearch(0, length, a, value));
}
```

Iterative Binary search

```
int binarySearch(int length, int a[], int value) {
  int left=0;
  int right = length;
  while (left < right - 1) {
    int mid = (left + right)/2;
    if (value < a[mid]){
      right = mid;
    } else {
      left = mid;
    }
  }
  if ((left < length) && (a[left] == value)) {
    return left;
  }
  return -1;
}
```

Binary search

Note that in each iteration of the searching process, the size of the search interval is halved.

Given input length n, we can do this $\log_2(n)$ times.

So, binary search has a logarithmic time complexity!

Comparison with linear search:
linar search needs (on average) 500.000 comparisons for a list of 1 million elements; binary search needs at most 20 comparisons for the ordered list with the same elements!

End week 6