Definition of An Algorithm

An algorithm is a way of solving a problem by:

- performing an *unambiguous* sequence of instructions in a finite amount of time and then halting.
- transforming input data into output data in finite time.

An algorithm can be considered as an abstraction of a computer program and may be implemented in any programming language.

Note that most graphs presented here are taken from the textbook slides.

Analysis of Algorithms

- The analysis of an algorithm should be done theoretically before it is implemented.
- We want to establish the algorithm efficiency using asymptotic notation approach
  - get a relation between the number of its basic operations and the size or magnitude of input data
  - provide an upper bound on the running time function.
- Proving the correctness of an algorithm using different proving techniques
  - A correct algorithm should return correct output for any acceptable input data, Also, it should halt after returning output.

Classification of Algorithms

- Generally, there are two types of algorithms: *iterative and recursive*.
- Classification of algorithms based on design techniques:
  1. divide and conquer
  2. dynamic programming
  3. greedy
  4. brute force
  5. backtracking
  6. branch and bound
  7. randomized

Classification of Algorithms (cont)

- *Brute Force* algorithms use non sophisticated approaches to solve a given problem. Typically, they are useful for small domains due to the total cost of examining all possible solutions.

  Example: sequential search of the sorted array or a Hamilton circuit.

- *Greedy* approach works by making a decision that seems the most promising at that moment and never reconsidering this decision. Greedy algorithms always choose a local optimum and only hope to end up with the global optimum.
  Examples: Dijkstra's, Kruskal's and Prim's algorithms for graphs, Huffman encoding for files compression, coin changing.

- *Divide and conquer* approach attempts to reduce a single large problem into multiple simple independent subproblems, solving the subproblems and then combining the solutions to these subproblems into the solution for the original problem.
  Examples: binary search, merge sort algorithms.

Dynamic Programming Algorithms

- *Dynamic programming* techniques are used for solving optimization problems by performing the following steps:
  - partitioning a problem into overlapping subproblems
  - recursively solving the subproblems and memoizing their solutions to avoid solving the same subproblems repeatedly
  - using an optimal structure approach to be sure that the optimal solutions of local subproblems lead to the optimal solution of the global problem.

  Two approaches are used: *top-down* and *bottom-up*. In general, the dynamic programming algorithm has better run times than the brute force algorithm.
  Examples: Fibonacci numbers, Floyd's all pairs shortest paths algorithm, matrix chain multiplication, longest common subsequence, activity scheduling problem.

Backtracking Algorithms

- *Backtracking* algorithm views the problem to be solved as a sequence of decisions and systematically considers all possible outcomes for each decision to solve the overall problem.
  - For example, it finds a solution to the first subproblem and then attempts to recursively solve the other subproblems based on this first solution. If it cannot, it backtracks and tries the next possible solution to the first subproblem and so on. Backtracking terminates when there are no more solutions to the first subproblem.
  - In some sense, backtracking algorithms are like brute-force algorithms. However, backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary and, therefore, it can perform much better .
  - An "intelligent backtracking" keeps track of the dependencies between sub-problems and only re-solves those which depend on an earlier solutions which have changed.

Examples: topological sort, Depth First Search, n-queens problem.

Other Important Algorithms

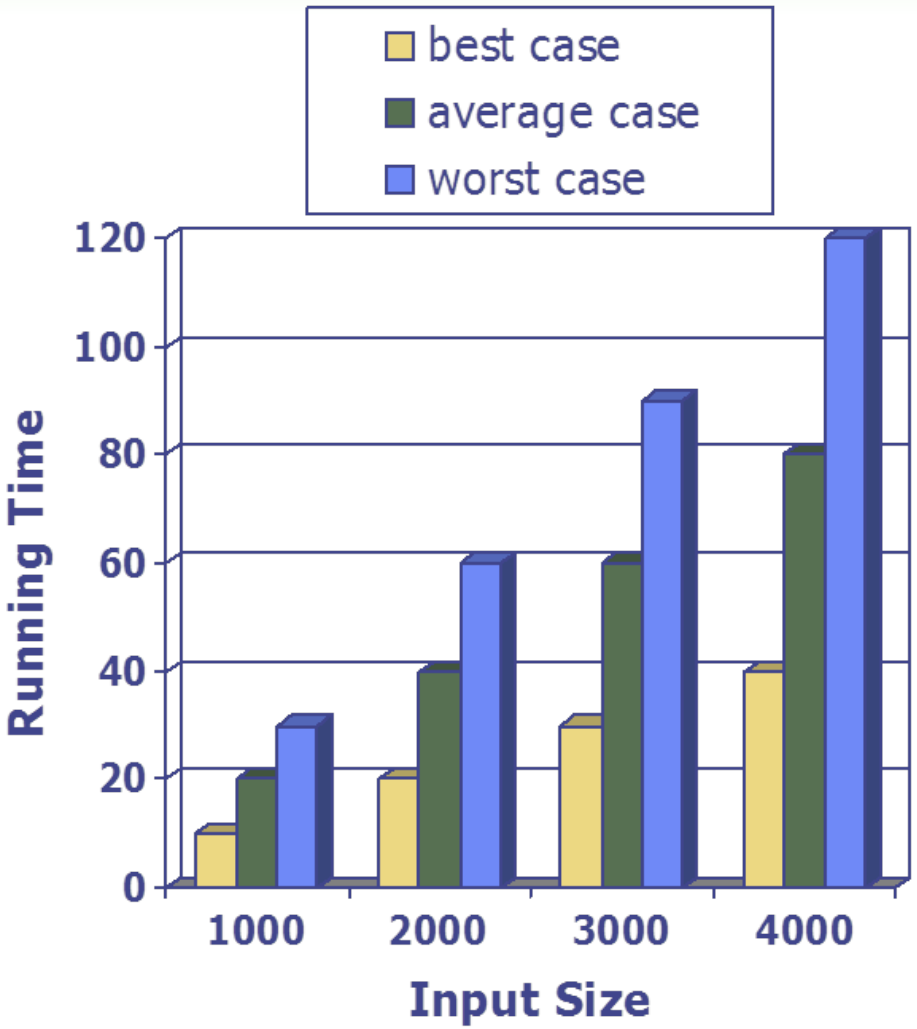
- *Branch and bound* algorithms find an optimal solution by keeping track of the best solutions found so far.
  - If a partial solution is not better than a current one then it is abandoned.
  - The algorithm traverses a spanning tree of the solution space and prunes the solution tree, thereby reducing the number of solutions to be considered.

  Examples: Linear programming and optimization problems.
- *Randomized* algorithms are algorithms that make some random (or pseudo-random) choices.
  Examples: randomized quick sort, pseudo-random number generator, probabilistic algorithms.

Running Time of an Algorithm

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- Therefore, we focus on the worst-case running time.
  - Easier to analyze
  - Crucial to applications such as scientific computations, games, finance and robotics

Typical Worst, Best and Average Cases

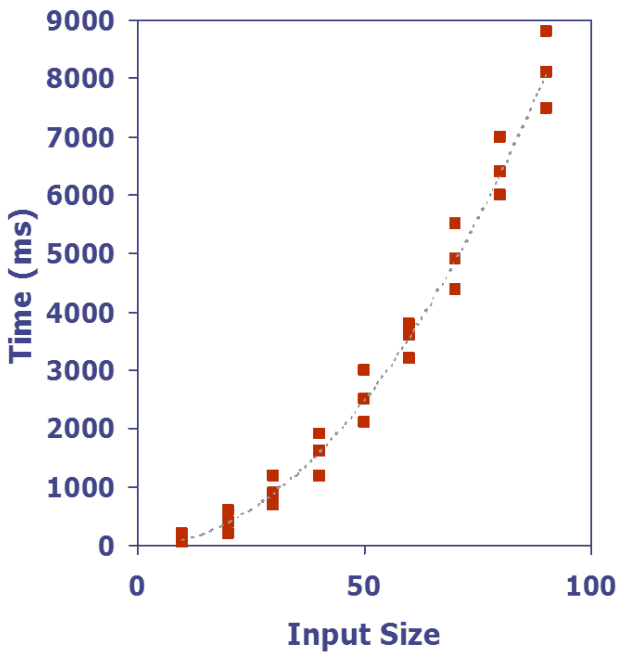


Why Do We Analyze Algorithms?

- We want to find answers to the following questions:
  - Does the algorithm work fast enough for my needs?
  - How much longer does it take when the input gets larger?
  - Which one of the several different algorithms is the fastest one?

Experimental Studies

- Write a program implementing an algorithm to be investigated.
- Run the program with inputs of varying sizes and compositions.
- Use a function, like the built-in `clock()` function, to get an accurate measure of the actual running time.
- Plot the run-time results and figure out a cost of the algorithm.



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running times on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

Theoretical Analysis

- Uses a *pseudocode*: a high-level description of an algorithm instead of an implementation in a particular programming language
  - More structured than English prose but less detailed than a program code
  - Preferred notation for describing algorithms
  - Hides some less important design issues like handling input and output
- Characterizes running time of an algorithm as a function of the input size, n.
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment.
- Takes into account all possible acceptable inputs (often implicitly).

Basic Operations

- The efficiency of an algorithm can be measured by how much of computer time and memory is utilized by its implementation.
- This can be studied in an abstract way without considering implementation details but focusing only on an algorithm's properties that affect its execution time.
- The efficiency analysis concentrates on basic operations:
  - data interchanges (swaps)
  - comparisons ($<, >, \leq, \geq, ==, !=$)
  - arithmetic operations ($+, *, -, /$,taking to power, square root, etc.)
- We assume that it takes a constant amount of time to access any cell in memory in the RAM (random-access machine) model.

Fact

Notice that this approach is independent of any programming language.

Growth Rate of Running Time Functions

- By inspecting the pseudocode, we can determine the maximum number of basic operations executed by an algorithm as a function of the input size.
- The number of input data items ($= n$) is directly connected with the number of operations performed by an algorithm and it is expressed as a *running time function*, $f(n)$.
- Usually, the running time function $f(n)$ is quite complex, so to classify an algorithm, we use its upper bound by a simpler function, $g(n)$, that expresses the growth rate.

Growth Rate of Running Time Functions (cont)

- Changing the hardware or software environment affects the running time by a positive constant factor C but it does not alter its growth rate.
- The number of operations of an algorithm is bounded by $Cg(n)$ and $g(n)$ is considered as a classification of the algorithm.
- A classification of an algorithm is done in terms of big-O asymptotic notation, denoted by $f(n) = O(g(n))$.

Basic Operations – `if` Statements

- The complexity (the number of operations) of `if` statements

```
if (condition) S1 else S2
```

- Total complexity = maximum of complexity of `S1` and complexity of `S2` which is not larger than complexity of `S1` plus complexity of `S2`.

Basic Operations – Loop Statements

- The complexity of a single loop:

```
int sum = 0;
for (int i = 1; i <= n; i++)
    sum = sum + 1;
```

- Amount of work of the algorithm = (the number of iterations) $\times$ (the number of basic operations at each iteration).
  - The number of operations for this loop: $f(n) = 2n + 1 = O(n)$.
- The complexity of a double loop:

```
int sum = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        sum = sum + 1;
```

  - The number of operations for this loop: $f(n) = 2n \cdot n + 1 = 2n^2 + 1 = O(n^2)$

Basic Operations – Loop Statements (cont)

- Another double loop:

```
int sum = 0; // one assignment
for (int i = 1; i <= n; i++)
   for (int j = 1; j <= i; j++)
      sum = sum + 1; // executed n(n+1)/2 times
```

  It requires $2n(n+1)/2+1 = O(n^2)$ operations.

- An equivalent result can be obtained without any loop:

```
sum = n*(n+1)/2;
// one assignment + one addition
// + one multiplication + one division.
```

  It requires 4 operations.

- Why $n+(n-1)+...+2+1 = n(n+1)/2$? You can use mathematical induction to prove it.

Linear Algorithms

- The linear running time functions:
  - Algorithm 1

```
int s = 0, t = 0;
for (int i = 1; i <= n; i++) s++;
for (int j = 1; j <= n; j++) t++;
```

    The number of operations: $f(n) = 2n + 2 = O(n)$
  - Algorithm 2

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

    The number of operations: $f(n) = \lceil n/2 \rceil + 1 = O(n)$

Logarithmic Algorithms

- The logarithmic running time functions:
  - Algorithm 1

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

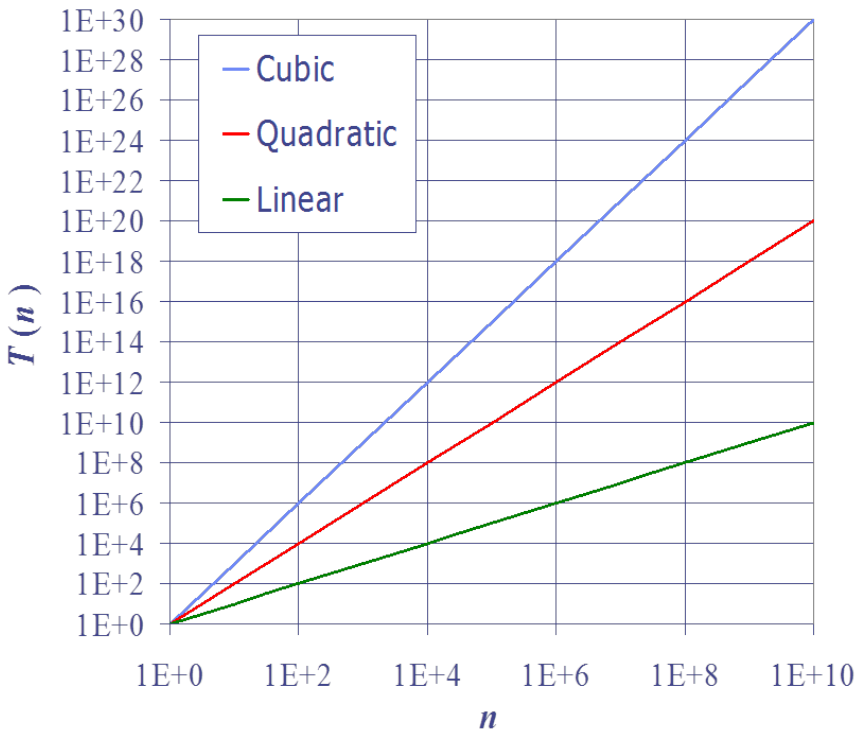
    The number of operations: $f(n) = 1 + \lfloor \log_2 n \rfloor + 1 = O(\log_2 n)$
  - Algorithm 2

```
int i = n;
while (i > 0)
    i = i/2;
```

    The number of operations:
    $f(n) = 2(\lfloor \log_2 n \rfloor + 1) + 1 = O(\log_2 n)$

Growth Rate Plots

- The growth rates could be expressed using *logarithmic scale*
- The slop of lines corresponds to the growth rates of running time functions
- The growth rates of running time functions are not affected by
  - constant factors (C, C_1, etc)
  - lower terms



Asymptotic Behavior of Algorithms

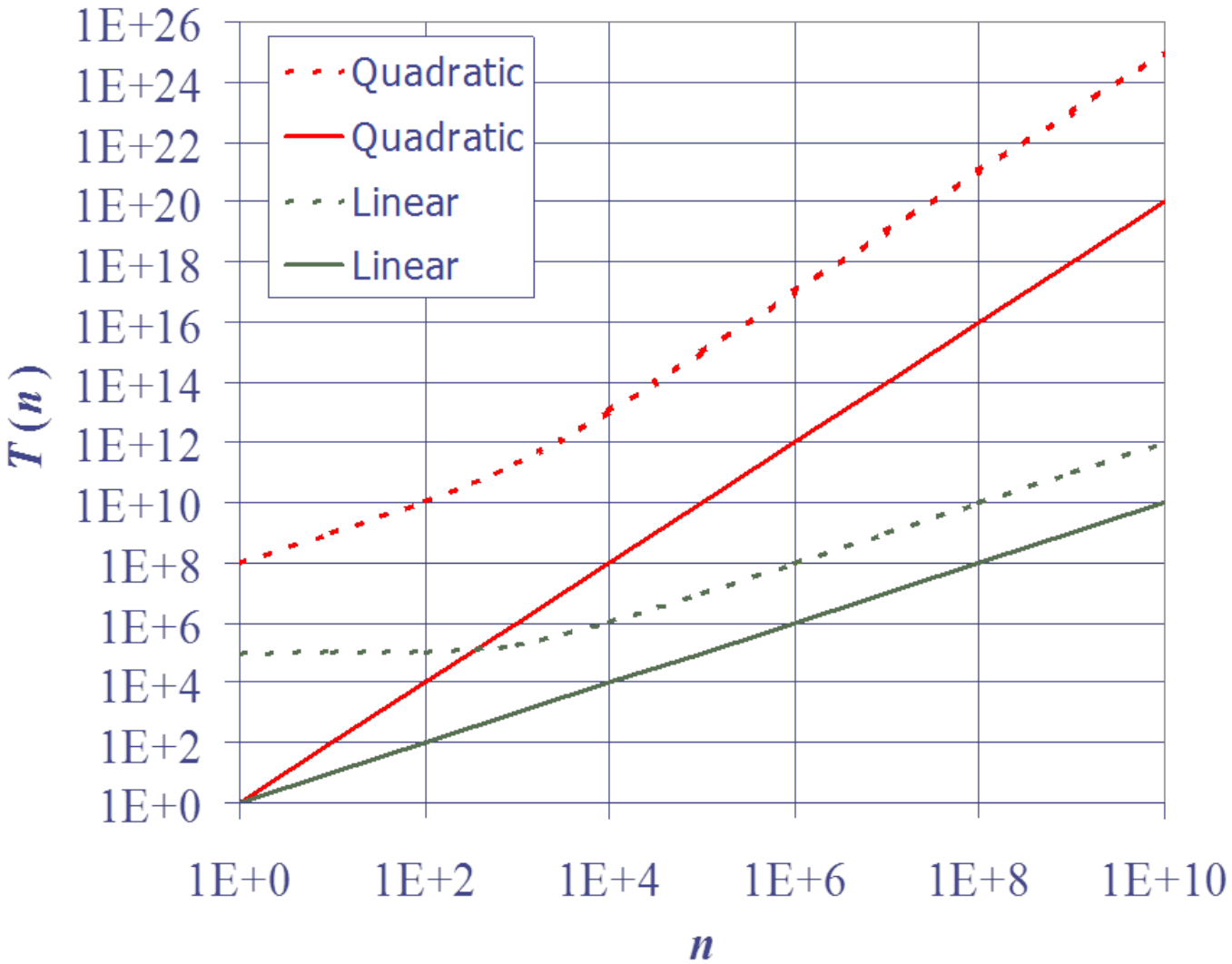
- The growth of a running time function is mainly defined by the highest order of n (where n = the size of input data) if n is large enough.
- For small n, most algorithms are very fast and we often do not care which algorithm we choose as long as it has at most a polynomial behavior (in contrast to exponential running time).
- From now on we are only interested in running times for large n. This will be called *asymptotic behavior*.

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

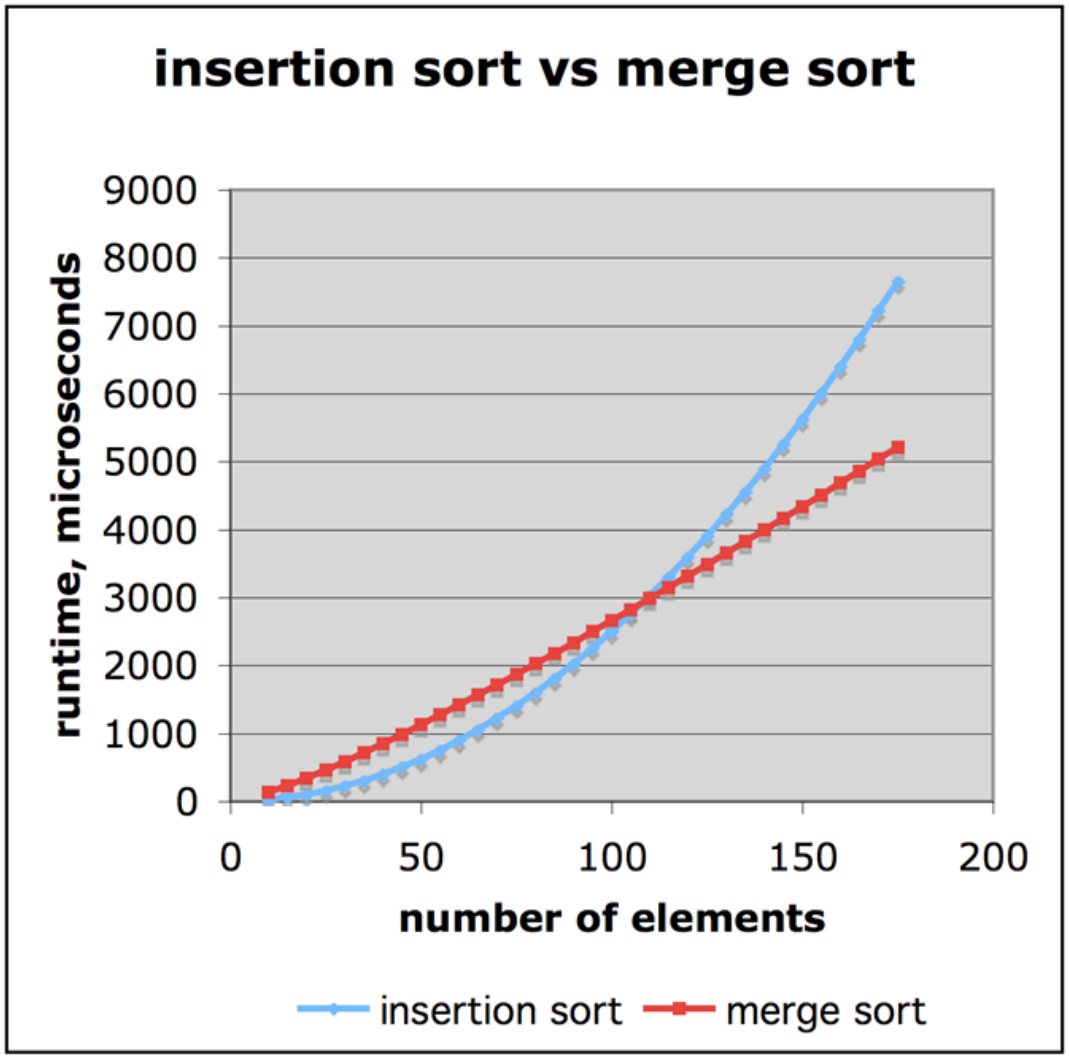
Examples

- Growth of a function is a characterization of an algorithm efficiency. For instance,
  - $f(n) = 4n^2 + 2n + 5$ has a quadratic rate of growth (the leading term is n^2)
  - $f(n) = 2n + 2$ has a linear rate of growth
  - $f(n) = 3\log_2(n) + 10$ has a logarithmic rate of growth

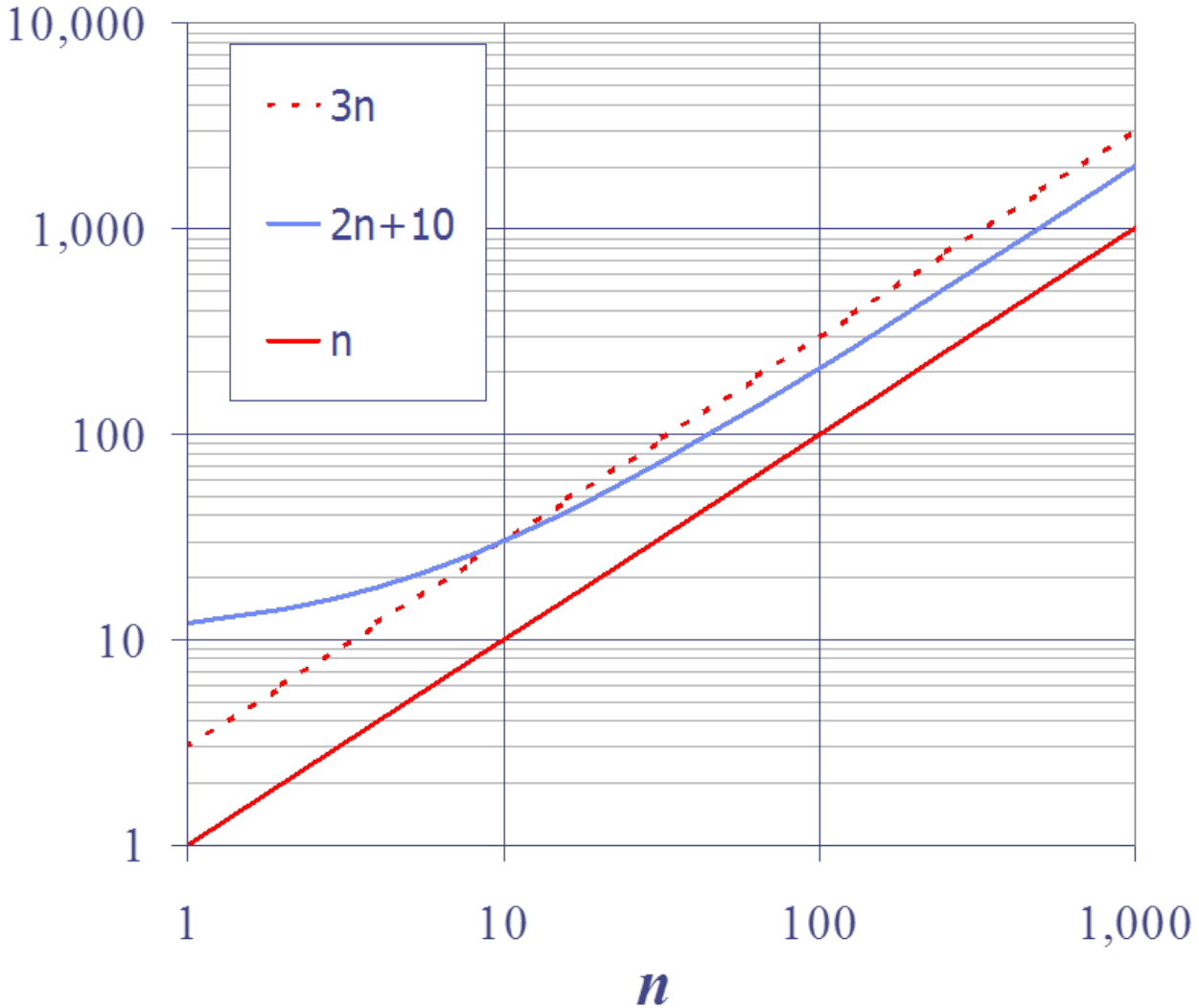
Constant Factors



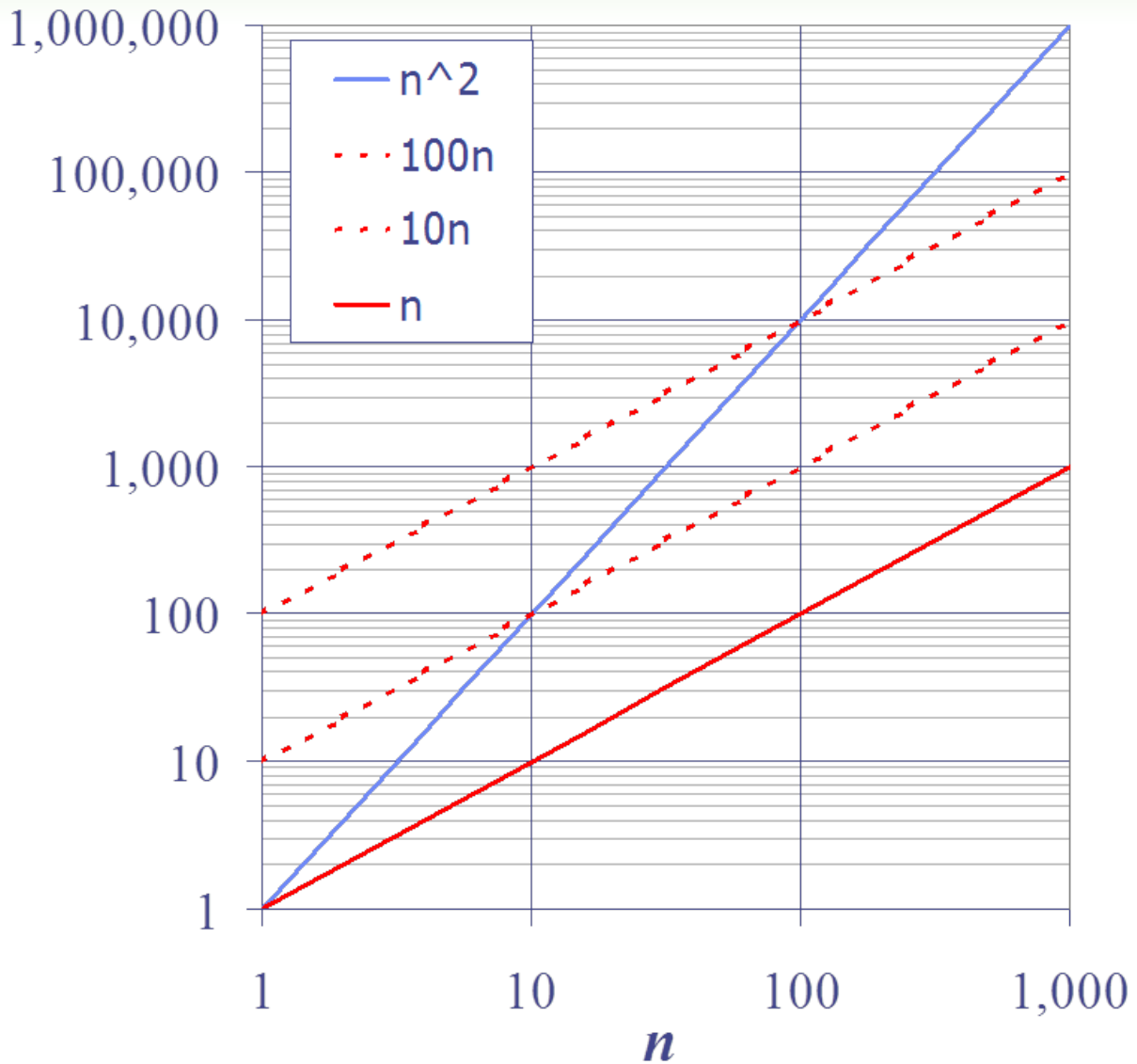
Comparison of Two Algorithms: Example 1



Comparison of Two Algorithms: Example 2



Comparison of Two Algorithms: Example 3



Types of Asymptotic Notation

- If the size of input grows, the running time function provides the asymptotic efficiency of an algorithm.
- The asymptotic analysis can be done using
  - Big-O (O) notation
  - Big-Omega (Ω) notation
  - Big-Theta (Θ) notation

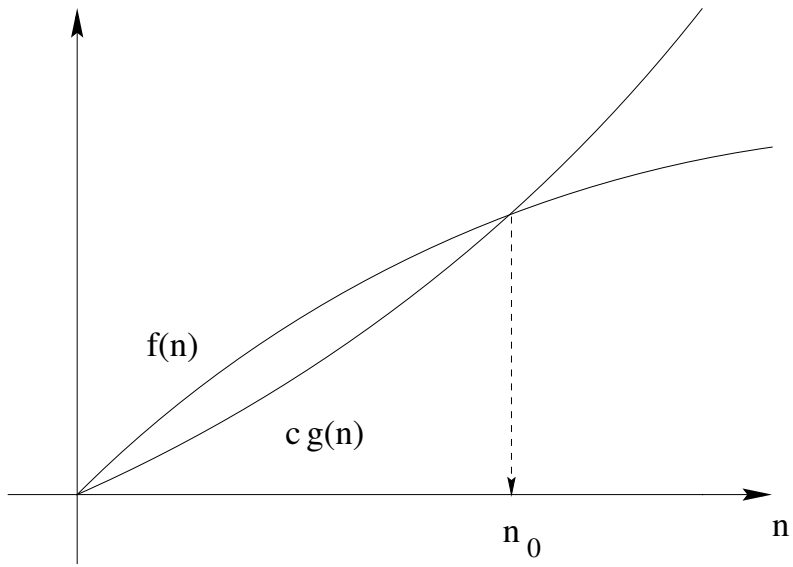
  as well as
  - little-o (o) notation
  - little-omega (ω) notation.

Big-O Notation

Definitions

1. *For a given function* $g(n)$ *by* $O(g(n))$ *we denote the set of functions* $O(g(n)) = \{f(n)$: *there exist positive constants* c *and* n_0 *such that* $0 \leq f(n) \leq cg(n)$ *for all* $n \geq n_0\}$.
2. $f(n) = O(g(n))$ *if and only if* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ *and* $c \geq 0$.

Big-O Notation (cont)



The function $cg(n)$ is the upper bound on $f(n)$ for almost all n. Note that for $n < n_0$ this inequality may or may not be valid, but it must be satisfied for any $n \geq n_0$.

Relatives of Big-O

- Big-Omega:
  $f(n) = \Omega(g(n))$ *if there exist constants* $c > 0$ *and* $n_0 \geq 1$ *such that* $f(n) \geq cg(n)$ *for* $n \geq n_0$.
- Big-Theta:
  $f(n) = \Theta(g(n))$ *if there exist constants* $c_1 > 0$, $c_2 > 0$ *and* $n_0 \geq 1$ *such that* $c_1 g(n) \leq f(n) \leq c_2 g(n)$ *for* $n \geq n_0$.
- Little-o:
  $f(n) = o(g(n))$ *if for any constant* $c > 0$, *there exists an integer constant* $n_0 \geq 0$ *such that* $f(n) \leq cg(n)$ *for* $n \geq n_0$.
- Little-omega:
  $f(n) = \omega(g(n))$ *if for any constant* $c > 0$, *there exists an integer constant* $n_0 \geq 0$ *such that* $f(n) \geq cg(n)$ *for* $n \geq n_0$.

Asymptotic Notation (cont)

Example

Let us consider the following algorithm:

```
int k = n;
while (k > 1)
    k = k/2;
```

The number of times that `k = k/2` is executed is $\lfloor \log_2 n \rfloor$.
Therefore, the runtime efficiency of this algorithm is $O(\log_2 n)$.

Asymptotic Notation (cont)

Examples

- If the number of operations of an algorithm is $n^3 + n^2 + 6n$ then such an algorithm is of order n^3, since

$$n^3 + n^2 + 6n \leq 3n^3 \qquad \text{if } n \geq 3,$$

  and we use notation $O(n^3)$.
- This doubly nested loop performs $O(n \log_2 n)$ arithmetic operations.

```
for (int j = 1; j <= n; ++j) { // n times
   int k = j;
   while (k >= 1) k = k/2; // O(lg n) times
}
```

Searching for a Good Algorithm

Example

PROBLEM TO SOLVE: *Compute the sum of* $1 + 2 + 3 + ... + n$ *for any integer* $n > 0$.

Algorithm 1:

```
1. sum = 0;  // one assignment
2. for i = 1 to n   // n iterations
3.     sum = sum + i; // 1 assignment + 1 addition
```

Algorithm 1 requires no more than $f(n) = (n+1)t_{=} + nt_{+}$ time units.

If $t = \max(t_{=}, t_{+})$ then $f(n) = (2n+1)t$ time units. Usually we drop time units and focus on the number of operations because they are platform and software independent. Time units do not change the growth rate of the function but only play the role of a scaling factor.

Searching for a Good Algorithm (cont)

Example

Algorithm 2

```
sum = 0; // one assignment
for (i = 1; i <= n; i++) {
   for (j = 1; j <= i; j++)
      sum = sum + 1; // n(n+1)/2 times
}
```

It requires $2(n(n+1)/2)+1 = O(n^2)$ operations. It computes the same result as Algorithm 1 but it is asymptotically slower.

Searching for a Good Algorithm (cont)

Example

Algorithm 3

```
sum = n*(n+1)/2;
// 1 assignment + 1 addition
// + 1 multiplication + 1 division.
```

It requires 4 operations.

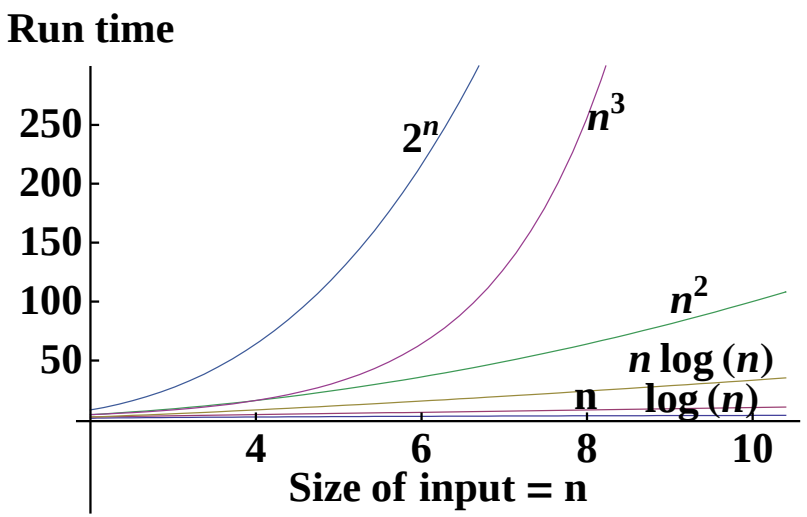
It computes the same result as Algorithm 1 and Algorithm 2 but it is asymptotically much faster than the two previous algorithms.

We used the fact that $n + (n-1) + (n-2) + ... + 1 = n(n+1)/2$.

You can use mathematical induction to prove it.

Classification of Algorithms According to Running Time Functions

1. constant $O(1)$
2. logarithmic $O(\log_2 n)$
3. linear $O(n)$
4. polynomial $O(n^k)$
5. exponential $O(2^n)$



Classification of Algorithms According to Running Time Functions (cont)

- The running time of the first four algorithms is bounded by polynomial functions and they are called **tractable** algorithms.
- Algorithms with polynomial running time are considered to be efficient and we could implement and run them for a reasonable large input data size.
- The last algorithm with the exponential running time function is an example of non-polynomial algorithm and it is called **non-tractable** algorithm.
- Exponential running times, like $O(a^n)$ for some constant $a > 1$, are not efficient for reasonably large input and take very long time to obtain results.

Classification of Algorithms According to Running Time Functions (cont)

Example

Consider five algorithms A_1, A_2, A_3, A_4, A_5 with running time functions: $f_1(n) = 10^n$, $f_2(n) = n^{1/3}$, $f_3(n) = n^n$,$f_4(n) = \log_2 n$, $f_5(n) = 2^{\log_2 \sqrt{n}}$, respectively.
List the algorithms from slowest to fastest.
Answer: $A3$, $A1$, $A2$, $A5$, $A4$

- To be able to compare these running time functions it is reasonable to take the logarithm of each function

	Notation: $z = \log_2 n$
A1: $f_1(n) = 10^n$	$\log_2 f_1(n) = n \log_2 10$
A2: $f_2(n) = n^{1/3}$	$\log_2 f_2(n) = z/3$
A3: $f_3(n) = n^n$	$\log_2 f_3(n) = nz$
A4: $f_4(n) = \log_2 n$	$\log_2 f_4(n) = \log_2 z$
A5: $f_5(n) = 2^{\log_2 \sqrt{n}}$	$\log_2 f_5(n) = z/2$

Predicted Running Times of Algorithms

- The table on the next slide presents an estimate on computational times of different algorithms on inputs of increasing size. We assumed that the processor performs a million (10^6) basic operations per second.

Input size	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	10^{-5} s	< 1 s	< 1 s	< 1 s	< 1 s	10^{-3}s	4 s
$n = 30$	$3 \cdot 10^{-5}$ s	< 1 s	< 1 s	< 1 s	< 1 s	18 m	10^{25} y
$n = 50$	$5 \cdot 10^{-5}$ s	< 1 s	< 1 s	< 1 s	11 m	36 y	∞
$n = 100$	10^{-4} s	< 1 s	< 1 s	1 s	$1.3 \cdot 10^4$ y	10^{17} y	∞
$n = 10^3$	10^{-3} s	< 1 s	1 s	18 m	∞	∞	∞
$n = 10^4$	10^{-2} s	< 1 s	2 m	12 d	∞	∞	∞
$n = 10^5$	10^{-1} s	2 s	3 h	32 y	∞	∞	∞
$n = 10^6$	1 s	20 s	12 d	$3.4 \cdot 10^4$ y	∞	∞	∞

Notation used: s = seconds, m = minutes, d = days, y = years, ∞ = very long

Example

- How to determine the function $f(n)$? Consider this example: *We want to multiply an $n \times n$ matrix A by a vector v and store the result in a vector u. How many operations does this algorithm need to perform?*
  - This algorithm performs $2n^2$ arithmetic operations (additions and multiplications), that is, $O(n^2)$ arithmetic operations.

```
void MatrixByVector(int n,
 double **A, double *v,
   double *u)
{
 int j, k;
 for (k = 0; k < n; k++) {
   u[k] = 0;
   for (j = 0; j < n; j++)
      u[k] += A[k][j]*v[j];
 }
}
```

- *How does the execution time of the algorithm depend on the size of the input data?*

Let $n = 10^4$ and assume that a computer performs one arithmetic operation in 10^{-9} second. Then multiplication of the matrix A by the vector v takes about 0.2 seconds.

But if we use the same algorithm with $n = 10^6$ then the execution time is about 33 minutes.

Complexity of Search Operation

Example

Consider the following linear search algorithm.

```
int lin_search(int A[], int n, int x) {
   for (int i = 0; i < n-1; i++)
      if (x == A[i]) return i;
   return -1;
}
```

Its running time depends on elements of the array `A`:

- The worst scenario is the longest running time of an algorithm on a certain input data. The above algorithm does n comparisons. It is the number of steps of the `for` loop to terminate, e.g., if the element x is not in the array.
- The random or average case: the number of comparisons of the algorithm is $n/2$. We expect to find the element in the middle of this array for some input.
- The best scenario: The input such that the first element of the array A is equal to x so the number of comparisons is equal to 1.

Binary Search

- Assume that our array is ordered by some relation (say $<$) and we search for a particular element called $\mathtt{x}$. How can the problem be solved?
  - The sequential (linear) search is one way to do it. Simply, we compare $\mathtt{x}$ with each element of the array until we have found the same element (see the previous slide). However, we do not use here information that the array is already sorted.
  - We want to develop some more efficient search strategy. We split the array according to some fixed guess. If the target $\mathtt{x}$ follows the data at the splitting location, we work with the data to the right of the splitting. If the target $\mathtt{x}$ precedes the data at the splitting location, we work with the data to the left of the splitting. This strategy is called *divide-and-conquer* strategy, and the algorithm itself is called the *binary search algorithm*.

Complexity of Binary Search

- This is a C++ implementation for the binary search algorithm.

```
int binary_search(int A[], int n, int x) {
    int mid, low = 0, high = n-1;
    while ( low <= high ) {
        mid = (low + high)/2;
        if ( A[mid] < x ) low = mid+1;
        else if( A[mid] > x ) high =  mid-1;
        else return mid; // found
    }
    return -1;  // not found
}
```

- The above algorithm uses two comparisons per iteration. For an unsuccessful search, the number of steps of the while loop is equal to $\lfloor \log_2 n \rfloor + 1$. For a successful search, the worst case is $\lfloor \log_2 n \rfloor$ iterations. The average case is $\lfloor \log_2 n \rfloor / 2$ iterations. Therefore, the binary search algorithm is of order $O(\log_2 n)$.

Sorting Algorithms

- Sorting is a process of rearranging a given set of objects (elements of an array or data in a sequential file) in a specific order using a specific relationship (corresponding to greater than or less than relations).
- Sorting is the preliminary step to a searching process. In everyday life we can see that objects are sorted in order to simplify searching, as for instance, in a dictionary, telephone book, or in a library.
- Sorting is an essential activity in data processing. Many algorithms are developed to sort data: *selection sort*, *insertion sort*, *bubble sort*, *Shell sort*, *quick sort*, *merge sort*, *heap sort*.
- A good measure of efficiency of these algorithms is obtained by computing the number of comparisons and swaps of objects. We will use the Big-O analysis to determine the number of operations.

Selection Sort

PROBLEM: Given a sequence of n integers. Sort it in ascending order.

Algorithm:

1. Set $k = 1$.
2. Select an index of the smallest object in the sequence of elements in positions from k to n.
3. Exchange the smallest object with the object in the k-th position.
4. Increase k by one.
5. Repeat the operations 2–4 until $k == n$.

We illustrate the selection sort by the following example.

6	4	8	10	1
1	4	8	10	6
1	4	8	10	6
1	4	6	10	8
1	4	6	8	10

Selection Sort Code

```
void selectionSort(int A[], int n)
{
    for (int k = 0; k < n-1; k++)
    {
        int index = k;
        for (int i = k+1; i < n; i++)
            if (A[i] < A[index]) index = i;
        int tmp = A[k]; // swap A[k] and A[index]
        A[k] = A[index];
        A[index] = tmp;
    }
}
```

Insertion Sort

This method is used by card players. Generally, if we have $k-1$ numbers in ascending order, we want to insert the k-th element in a proper place in order to preserve the order. We demonstrate the procedure in the following example:

6	4	10	8	1
4	6	10	8	1
4	6	10	8	1
4	6	8	10	1
1	4	6	8	10

Insertion Sort Code

```
void insertionSort(int A[], int n)
{
    for (int k = 1; k < n; k++ )
    {
        int tmp = A[k];
        int j = k;
        for( ; j > 0 && tmp < A[j-1]; j--)
            A[j] = A[j-1];
        A[j] = tmp;
    }
}
```

Efficiency of Insertion Sort

- The number of comparisons and swaps (three assignments) depends on initial order of elements of an array `A`.
- The best case is when the sequence is already sorted; then we have $n - 1$ comparisons and no swaps.
- The worst case is when the sequence is in the reverse order; we have $(n^2 - n)/2$ comparisons and swaps.
- In the average case we have $(n^2 - n)/4$ comparisons and swaps. To analyze the average case we use the probability approach. The number of comparisons and swaps are of the same order.

Bubble Sort

- The bubble sort is illustrated by this example.

6	4	8	10	1
4	6	8	10	1
4	6	8	1	10
4	6	1	8	10
4	1	6	8	10
1	4	6	8	10

  The largest element of the sequence goes to the last position for each subsequence of k numbers.

- The similar analysis as for the insertion sort can be applied.

Bubble Sort Code

```
void bubbleSort(int A[], int n)
{
   for (int k = 1; k < n; k++) {
      bool cont = false;
      for (int j = 0; j < n - k; j++)
         if (A[j+1] < A[j]) {
            int tmp = A[j]; // swap A[j] and A[j+1]
            A[j] = A[j+1];
            A[j+1] = tmp;
            cont = true;
         }
         if (!cont) break; // stop sorting
   }
}
```

Shell Sort

- Consider the following example: $5, 8, 2, 6, 1, 9, 3, 7, 4$.
  - We divide this array into 3 segments: $5, 6, 3||8, 1, 7||2, 9, 4$ and sort each segment using insertion sort: $3, 5, 6||1, 7, 8||2, 4, 9$.
  - The original array is now partially sorted: $3, 1, 2, 5, 7, 4, 6, 8, 9$.
  - We divide this array into 2 segments: $3, 2, 7, 6, 9||1, 5, 4, 8$ and we sort each segment using insertion sort: $2, 3, 6, 7, 9||1, 4, 5, 8$.
  - The array takes the form: $2, 1, 3, 4, 6, 5, 7, 8, 9$.
  - We sort this array as a whole using insertion sort: $1, 2, 3, 4, 5, 6, 7, 8, 9$.

Shell Sort (cont)

- The general idea of the Shell algorithm is such that the whole array is first fragmented into segments whose elements are a distance `gap` apart. The number of such segments is equal to `gap`, and the segments are as follows (here k = `gap`):

A[1]	A[k+1]	A[2*k+1]	...
A[2]	A[k+2]	A[2*k+3]	...
...	...	...	...
A[k]	A[2*k]	A[3*k]	...

- The idea of this algorithm is based on sorting of subarrays instead of sorting the whole array.

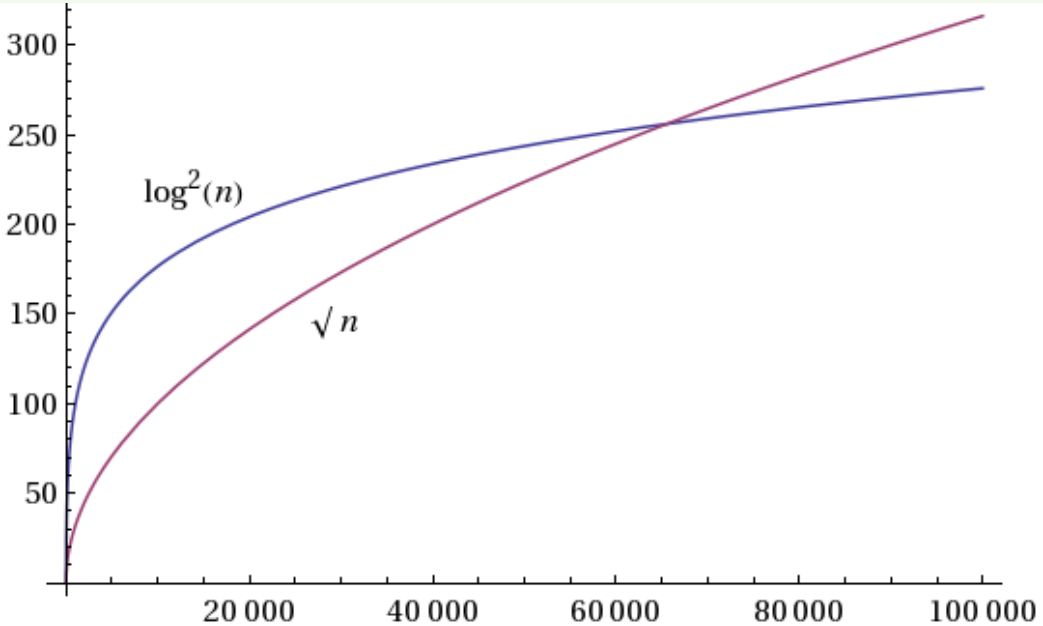
Shell Sort Code

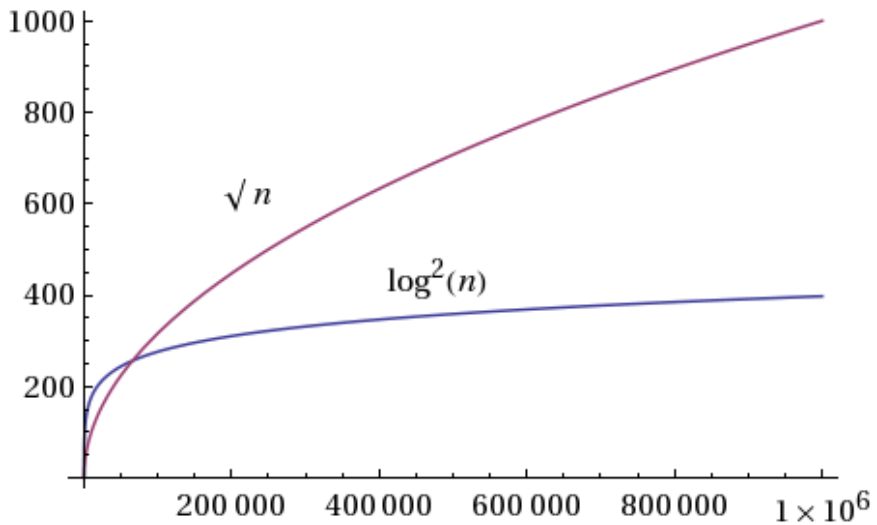
```
inline int next_gap(int gap)
{
   if (gap == 2) return 1;
   else return (int) (gap/2.2);
}
void shellSort(int A[], int n)
{
   int gap = n/2;
   for (; gap > 0; gap = next_gap(gap)) {
      for (int i = gap; i < n; i++) {
         int tmp = A[i];
         int j = i;
         while (j >= gap && tmp < A[j-gap]) {
            A[j] = A[j-gap];
            j -= gap;
         }
         A[j] = tmp;
      }
   }
}
```

Complexity of Shell Sort

- Because each segment is sorted, the whole array is partially sorted after the first partition. Next, the value `gap` is reduced and as a result the size of segments is increased. A good value of `gap` is such that is relatively prime to its previous value. The process reaches the value 0 for `gap`, and the array is sorted. The insertion sort is applied to each segment, so each successive segment is partially sorted.
- From the algorithm implementation we see that there are three loops. Let $k =$ `gap`. The outer `for` loop in the `shellSort` procedure needs $O(\log_2 n)$ repetitions. The mid `for` loop needs $n - k$ repetitions for each k. It is rather difficult to predict how many iterations are in the inner `while` loop; it depends on the order of data within the segment.
- From computational experiments we obtain that the average number of comparisons is between $O(n(\log_2 n)^2)$ and $O(n^{3/2})$.

Comparison of Running-Time Functions





Big-Omega Notation

Ω notation provides an (asymptotic) lower bound of a running time function.

Definitions

1. *For a given function* $g(n)$ *we denote by* $\Omega(g(n))$ *the set of functions* $\Omega(g(n)) = \{f(n)$: *there exist positive constants* c *and* n_0 *such that* $0 \leq cg(n) \leq f(n)$ *for all* $n \geq n_0$ $\}$.
2. $f(n) = \Omega(g(n))$ *iff* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ *and* $0 < c \leq \infty$.

Examples

- $2n^3 - 3n^2 + 2 = \Omega(n^3)$
- $n + 2\log_2 n - 3 = \Omega(n)$
- $a^n = \Omega(n)$ and $a > 1$

Theta Notation

Θ notation provides an asymptotic tight (upper and lower) bound of a running time function.

Definitions

1. For a given function $g(n)$ by $\Theta(g(n))$ we denote the set of functions $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.
2. $f(n) = \Theta(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ and $0 < c < \infty$

Example

$\sum_{k=1}^{n} k^2 = \frac{1}{6} n(n+1)(2n+1) = \Theta(n^3)$

Theorem

For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Asymptotic Notation (cont)

- If we have Big-O and Big-Omega bounds then we can easily get a Big-Theta bound.
  - Often the Ω bound is a bound of the best-case running time, and the O bound is a bound of the worst case running time of an algorithm.

Problems

1. Show that $f(n) = an^2 + bn + c = \Theta(n^2)$ with $c_1 = a/4$ and $c_2 = 7a/4$ for some n_0.
2. Show that for any polynomial in n of degree d

$$\sum_{i=0}^{d} a_i n^i = \Theta(n^d)$$

assuming that $a_d > 0$.

Little-o Notation

Little-o notation is an upper bound of a running time function, but it is not asymptotically tight.

Definition

For a given function $g(n)$ by $o(g(n))$ we denote the set of functions $o(g(n)) = \{f(n)$: for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Theorem

The relation $f(n) = o(g(n))$ *implies* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Examples

- $2n = o(n^2)$ but $2n^2 \neq o(n^2)$
- $n^b = O(a^n)$ for $a > 1$ and $n^b = o(a^n)$

Little-omega Notation

ω notation provides a lower bound of a running time function, which is not asymptotically tight.

Definition

For a given function $g(n)$ by $\omega(g(n))$ we denote the set of functions $\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

Theorem

The relation $f(n) = \omega(g(n))$ *implies* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Examples

- $n^2 = \omega(n)$ but $n \neq \omega(n)$
- $a^n = \omega(n)$ and $a > 1$

Properties

- **Transitivity**
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
  - $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$
  - $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$
  - $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$
  - $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$
- **Reflexivity**
  - $f(n) = \Theta(f(n))$
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$

Properties (cont)

- **Symmetry**
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- **Transpose symmetry**
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
  - $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$
- **Linearity**

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta(\sum_{k=1}^{n} f(k))$$