T&R Team of Algorithm Design
College of Computer Science and Engineering, CQU

Algorithm Analysis & Design

Introduction to Algorithm

Chapter 3: Asymptotic Analysis for Algorithms

Outline

- **3.1 ASYMPTOTIC ANALYSIS & LANDAU SYMBOLS**

- **3.2 ANALYSIS of OPERATIONS**

3.1 ASYMPTOTIC ANALYSIS & LANDAU SYMBOLS

Background

Background

- **Suppose we have two algorithms, how can we tell which is better?**

Background

- **Suppose we have two algorithms, how can we tell which is better?**

- **We could implement both algorithms, run them both**

Background

- **Suppose we have two algorithms, how can we tell which is better?**

- **We could implement both algorithms, run them both**
  - **Expensive and error prone**

Background

- **Suppose we have two algorithms, how can we tell which is better?**

- **We could implement both algorithms, run them both**
  - **Expensive and error prone**

- **Preferably, we should analyze them mathematically**

Background

- **Suppose we have two algorithms, how can we tell which is better?**

- **We could implement both algorithms, run them both**
  - **Expensive and error prone**

- **Preferably, we should analyze them mathematically**
  - **Algorithm analysis**

Background

Background

- **Without algorithm analysis, there will always be lingering questions:**

Background

- **Without algorithm analysis, there will always be lingering questions:**
  - **Was the algorithm implemented correctly?**

Background

- **Without algorithm analysis, there will always be lingering questions:**
  - **Was the algorithm implemented correctly?**
  - **Are there any bugs?**

Background

- **Without algorithm analysis, there will always be lingering questions:**
  - **Was the algorithm implemented correctly?**
  - **Are there any bugs?**
  - **How much faster?**

Background

Background

- **You may have heard that on your work-term reports, you should use quantitative analysis instead of qualitative analysis**

Background

- **You may have heard that on your work-term reports, you should use quantitative analysis instead of qualitative analysis**

- **The second refers to comparison of qualities, e.g., faster, less memory, etc.**

Background

- **You may have heard that on your work-term reports, you should use quantitative analysis instead of qualitative analysis**

- **The second refers to comparison of qualities, e.g., faster, less memory, etc.**

- **Engineers must determine the actual costs (memory, time, monetary) involved with the algorithms they propose**

Asymptotic Analysis

Asymptotic Analysis

- **In general, we will always analyze algorithms with respect to one or more variables**

Asymptotic Analysis

- **In general, we will always analyze algorithms with respect to one or more variables**

- **We will begin with one variable:**

Asymptotic Analysis

- **In general, we will always analyze algorithms with respect to one or more variables**

- **We will begin with one variable:**
  - **The number of items n currently stored in an array or other data structure**
  - **The number of items expected to be stored in an array or other data structure**
  - **The dimensions of an $n \times n$ matrix**

Asymptotic Analysis

- **In general, we will always analyze algorithms with respect to one or more variables**

- **We will begin with one variable:**
  - **The number of items n currently stored in an array or other data structure**
  - **The number of items expected to be stored in an array or other data structure**
  - **The dimensions of an $n \times n$ matrix**

- **Examples with multiple variables:**

Asymptotic Analysis

- **In general, we will always analyze algorithms with respect to one or more variables**

- **We will begin with one variable:**
  - **The number of items n currently stored in an array or other data structure**
  - **The number of items expected to be stored in an array or other data structure**
  - **The dimensions of an $n \times n$ matrix**

- **Examples with multiple variables:**
  - **Dealing with n objects stored in m memory locations**
  - **Multiplying a $k \times m$ and an $m \times n$ matrix**
  - **Dealing with sparse matrices of size $n \times n$ with m non-zero entries**

Find the Key Index

Find the Key Index

- **For example, we can check the required time to search for a certain value in a sorted, *n*-element-array**

Find the Key Index

- **For example, we can check the required time to search for a certain value in a sorted, *n*-element-array**

- **We first apply the linear search**

```
int find_keyIndex( int *array, int n, int key) {

    int result = -1;

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] == key) {
            result = i;
            break;
        }
    }

    return result ;
}
```

Find the Key Index

- **For example, we can check the required time to search for a certain value in a sorted, *n*-element-array**

- **We first apply the linear search**

```
int find_keyIndex( int *array, int n, int key) {

    int result = -1;

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] == key) {
            result = i;
            break;
        }
    }

    return result ;
}
```

The average comparison count for linear search would be *n*/2

Find the Key Index

Find the Key Index

- **And then we apply the binary search**

```
int binary_search( int *a, int n, int key ) {

   int mid, front=0, back=n-1;

   while (front<=back) {

       mid = (front+back)/2;

       if (a[mid]==key)
           return mid;

       if (a[mid]<key)
           front = mid+1;
       else
           back = mid-1;
   }

   return -1;
}
```

Find the Key Index

- **And then we apply the binary search**

```
int binary_search( int *a, int n, int key ) {

   int mid, front=0, back=n-1;

   while (front<=back) {

       mid = (front+back)/2;

       if (a[mid]==key)
           return mid;

       if (a[mid]<key)
           front = mid+1;
       else
           back = mid-1;
   }

   return -1;
}
```

The average comparison count for binary search would be log(*n*/2)

Linear and Binary Search

n

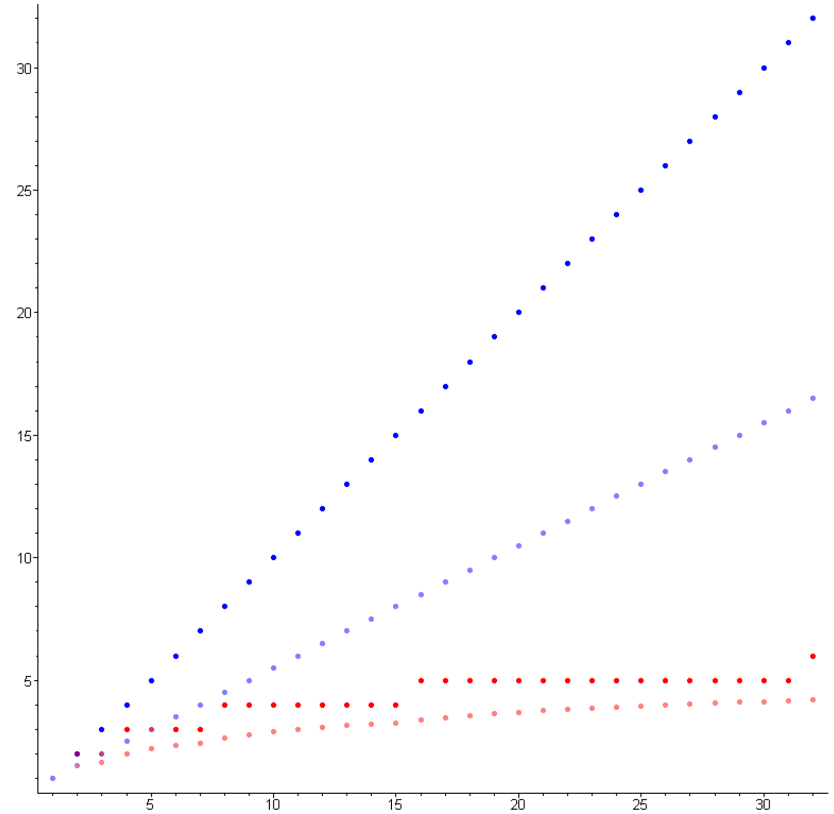
Linear and Binary Search

- **This plot shows maximum and average number of comparisons to find an entry in a sorted array of size *n***

n

Linear and Binary Search

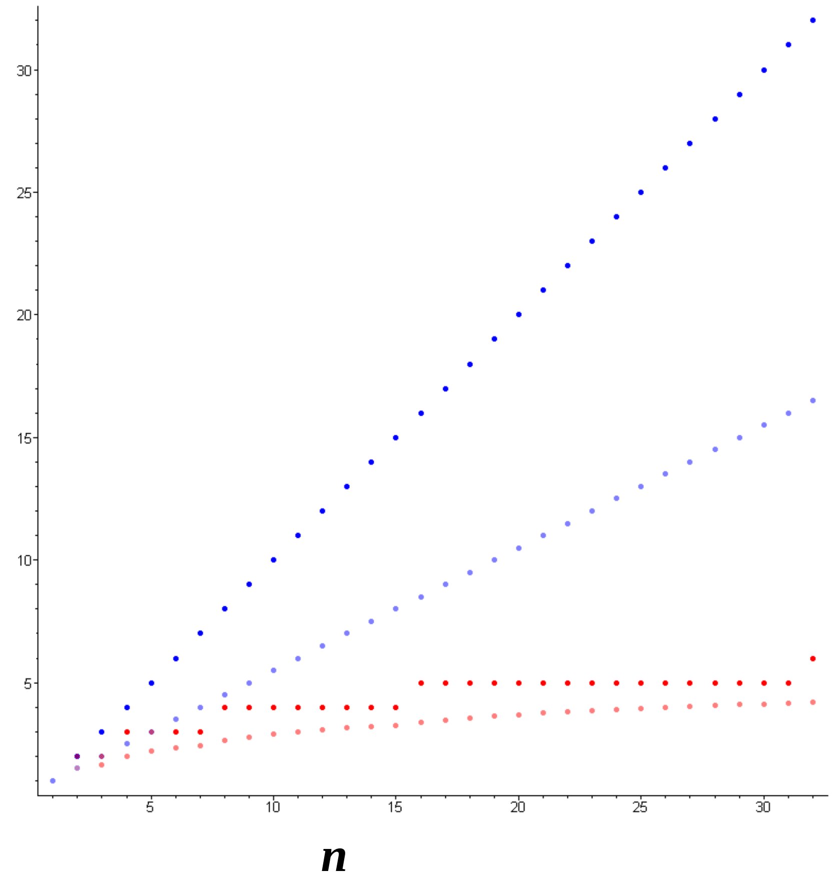
- **This plot shows maximum and average number of comparisons to find an entry in a sorted array of size *n***
  - **Linear search**
  - **Binary search**



n

Linear and Binary Search

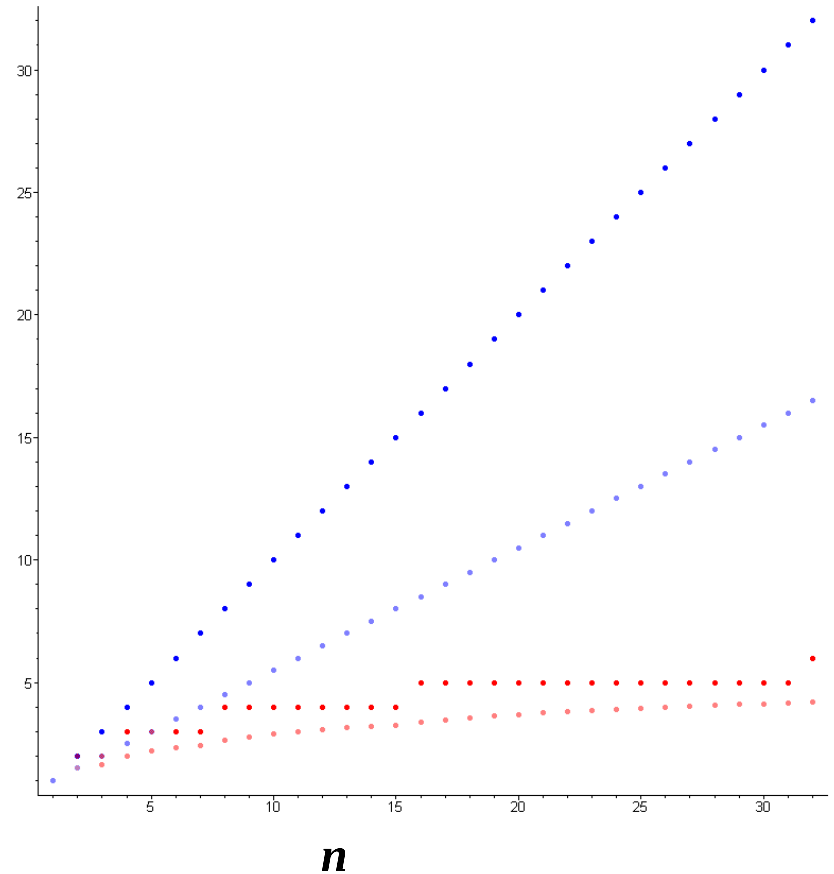
- **This plot shows maximum and average number of comparisons to find an entry in a sorted array of size *n***
  - **Linear search**
  - **Binary search**



- **We see than the growth of linear search and binary search are very different.**

Linear and Binary Search

- **This plot shows maximum and average number of comparisons to find an entry in a sorted array of size *n***
  - **Linear search**
  - **Binary search**



- **We see than the growth of linear search and binary search are very different.**

- **How about the functions with the same order?**

Quadratic Growth

Quadratic Growth

- **Consider the two functions**

$$\mathbf{f}(n) = n^2$$

$$\mathbf{g}(n) = n^2 - 3n + 2$$

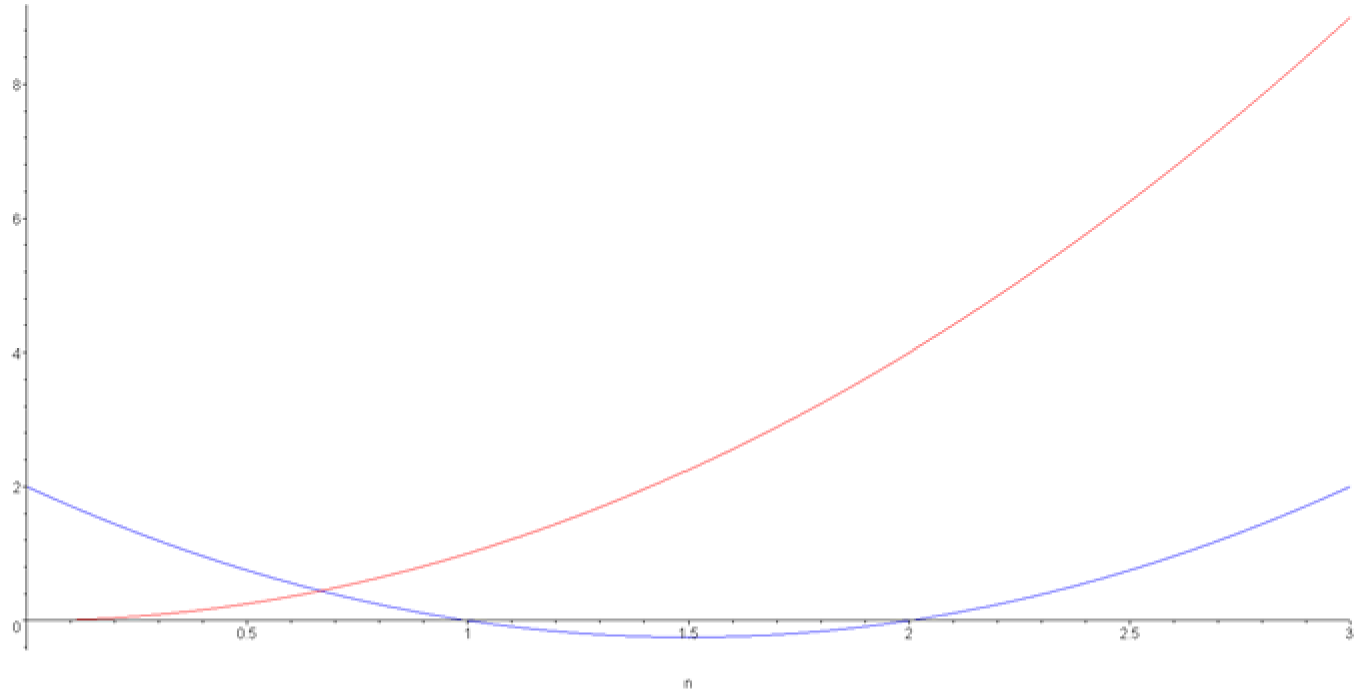
Quadratic Growth

- **Consider the two functions**

$$\mathbf{f}(n) = n^2$$

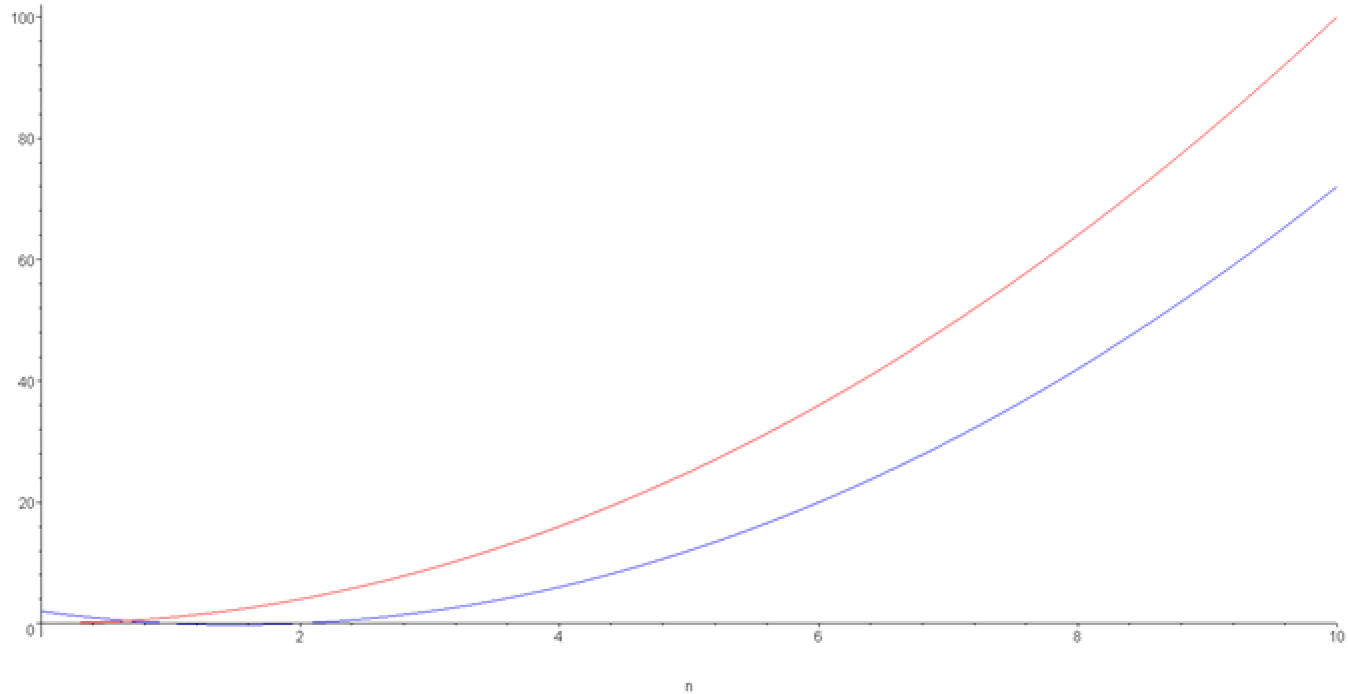
$$\mathbf{g}(n) = n^2 - 3n + 2$$

- **Around n = 0, they look very different**



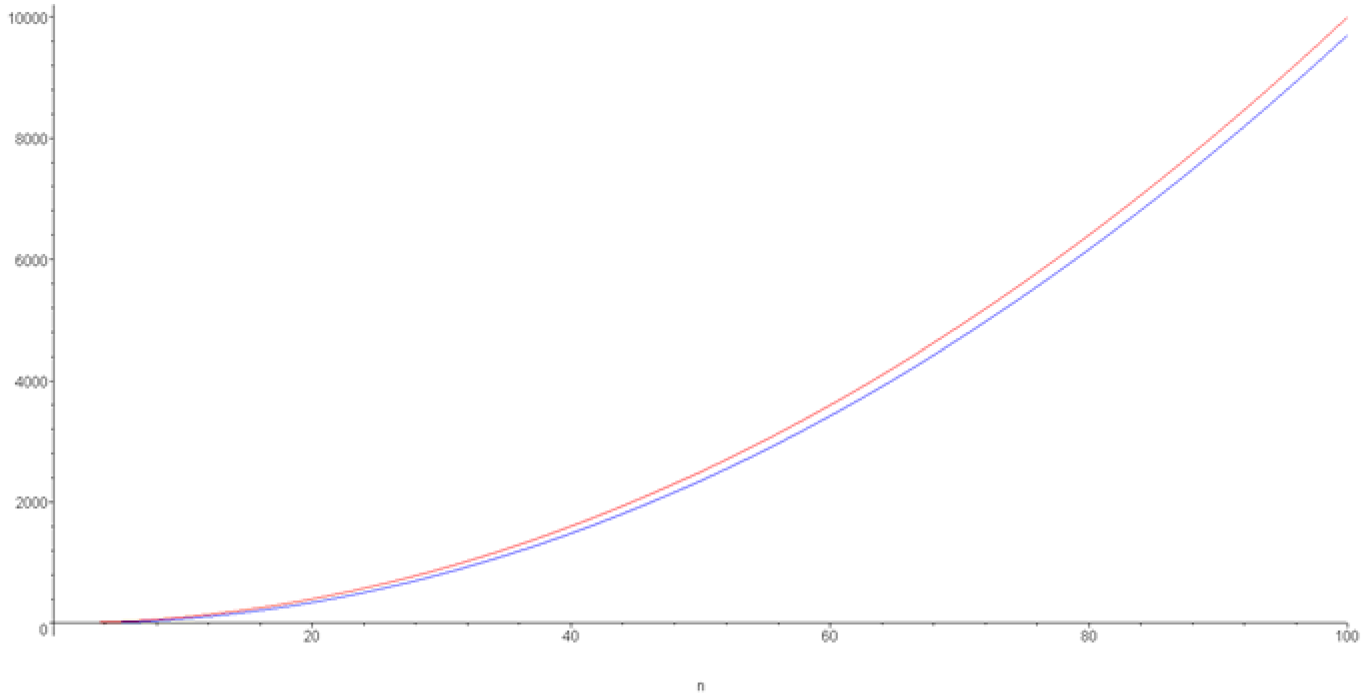
Quadratic Growth

- If we look at a slightly larger range from *n* = [0, 10], we begin to note that they are more similar:



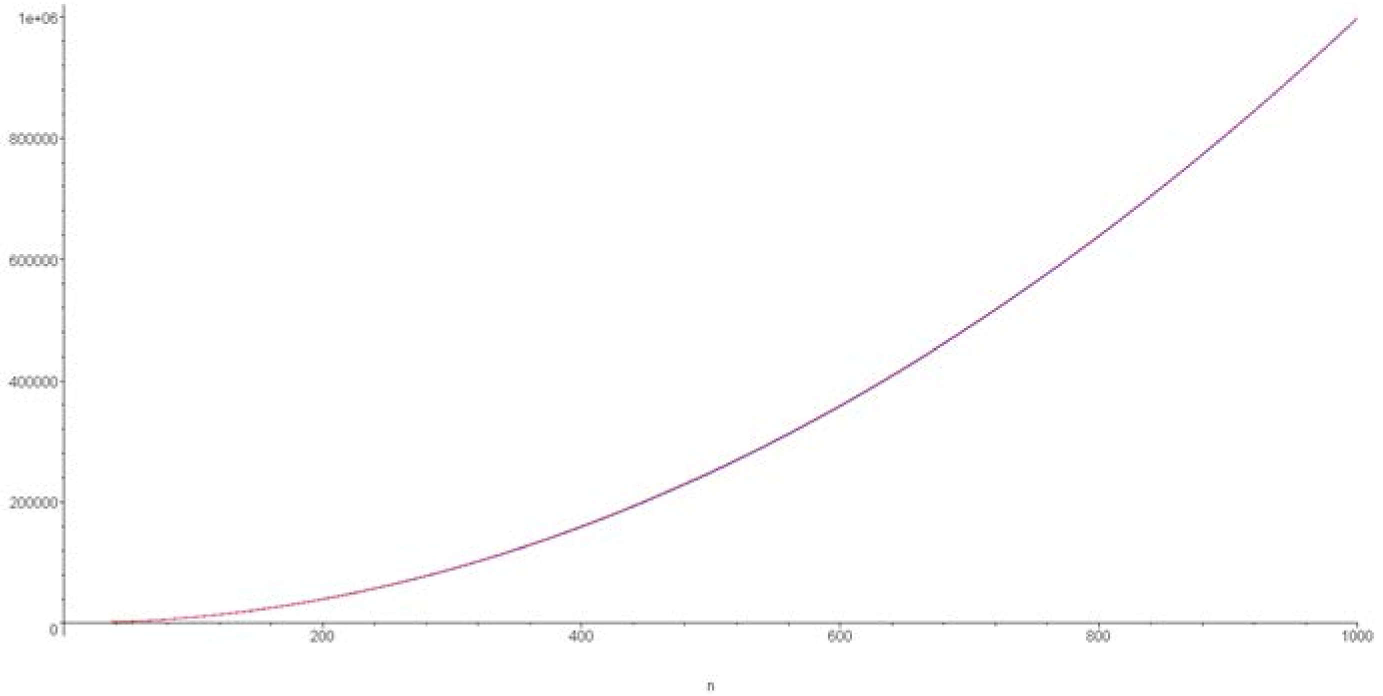
Quadratic Growth

- **Extending the range to *n* = [0, 100], the similarity increases:**



Quadratic Growth

- **And on the range *n* = [0, 1000], they are (relatively) indistinguishable:**



Quadratic Growth

- **The are different absolutely, for example,**

  f(1000) = 1 000 000

  g(1000) = 997 002

Quadratic Growth

- **The are different absolutely, for example,**

  f(1000) = 1 000 000

  g(1000) = 997 002

- **However, this relative difference is very small:**

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

Quadratic Growth

- **The are different absolutely, for example,**

  f(1000) = 1 000 000

  g(1000) = 997 002

- **However, this relative difference is very small:**

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

- **And this difference goes to zero as $n \to \infty$**

Polynomial Growth

Polynomial Growth

- **To demonstrate with another example,**

  $$\mathbf{f}(n) = n^6$$

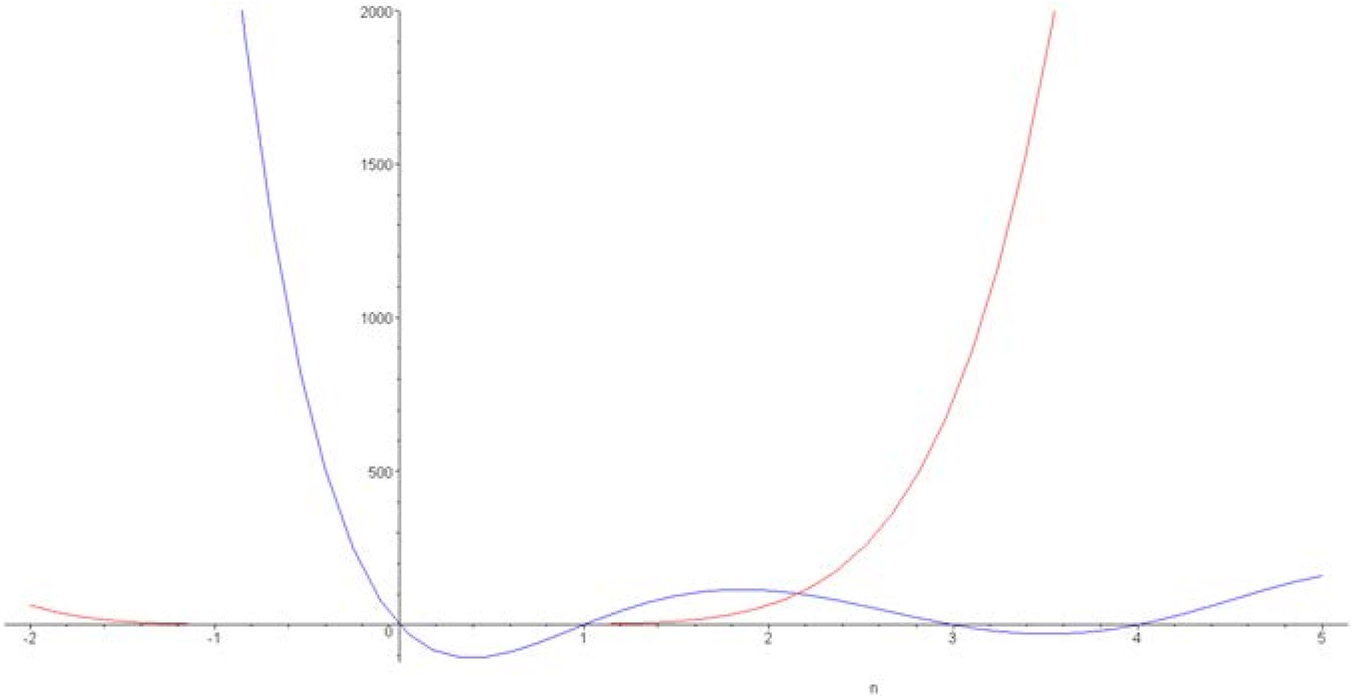
  $$\mathbf{g}(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$$

Polynomial Growth

- To demonstrate with another example,

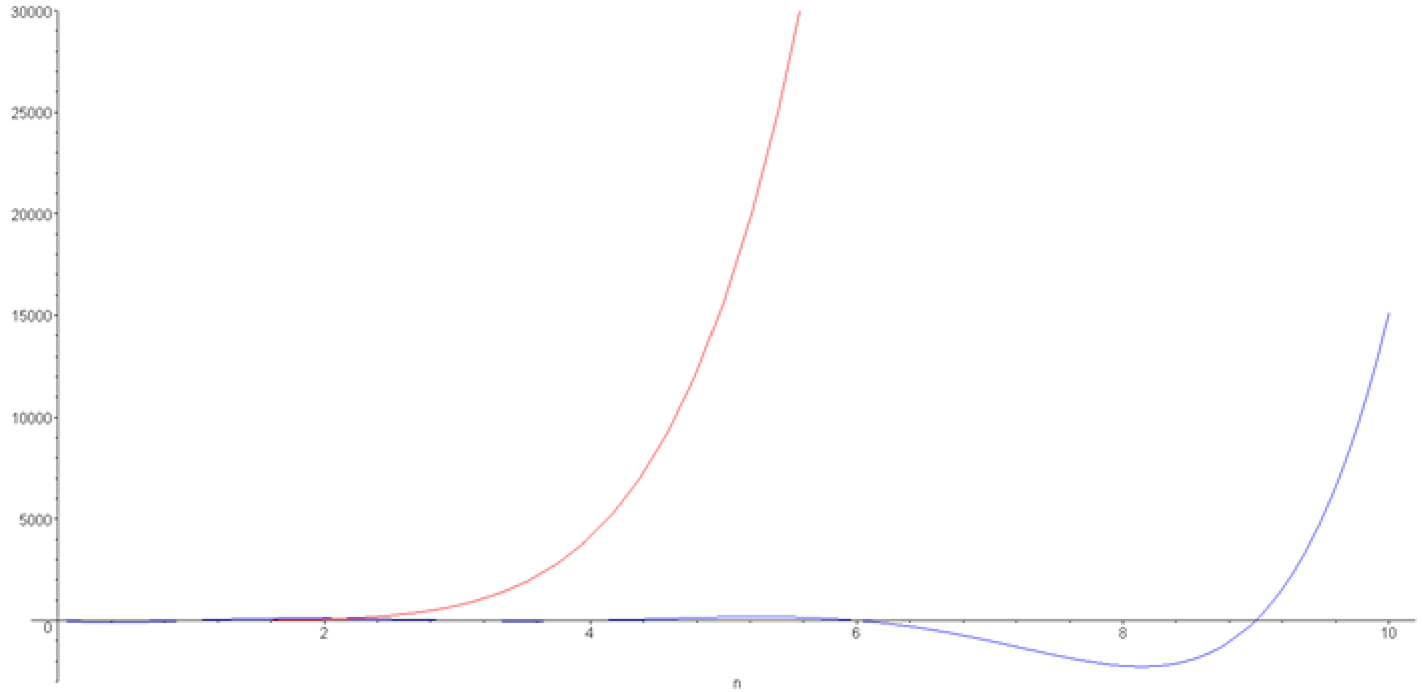
  $f(n) = n^6$

  $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$
- Around $n = 0$, they are very different



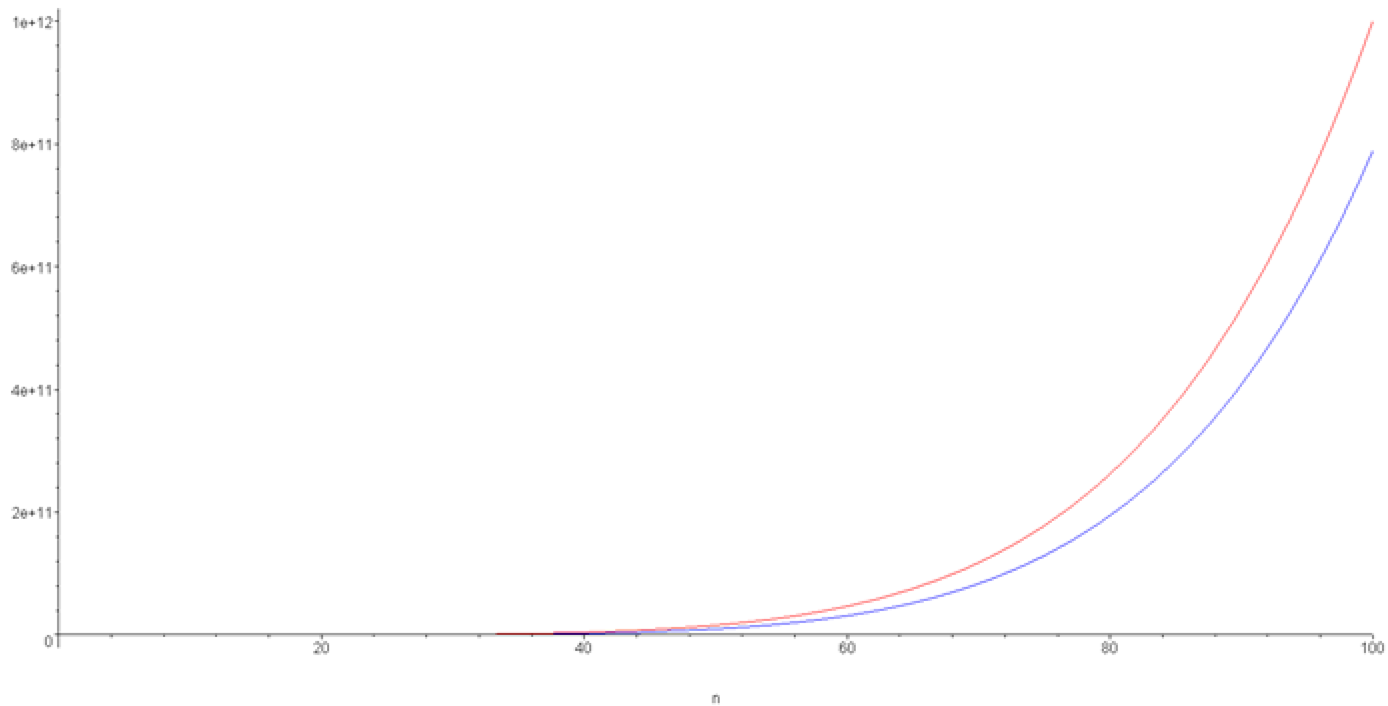
Polynomial Growth

- **Even extending the range to *n* = [0, 10] does not appear to give much similarity**



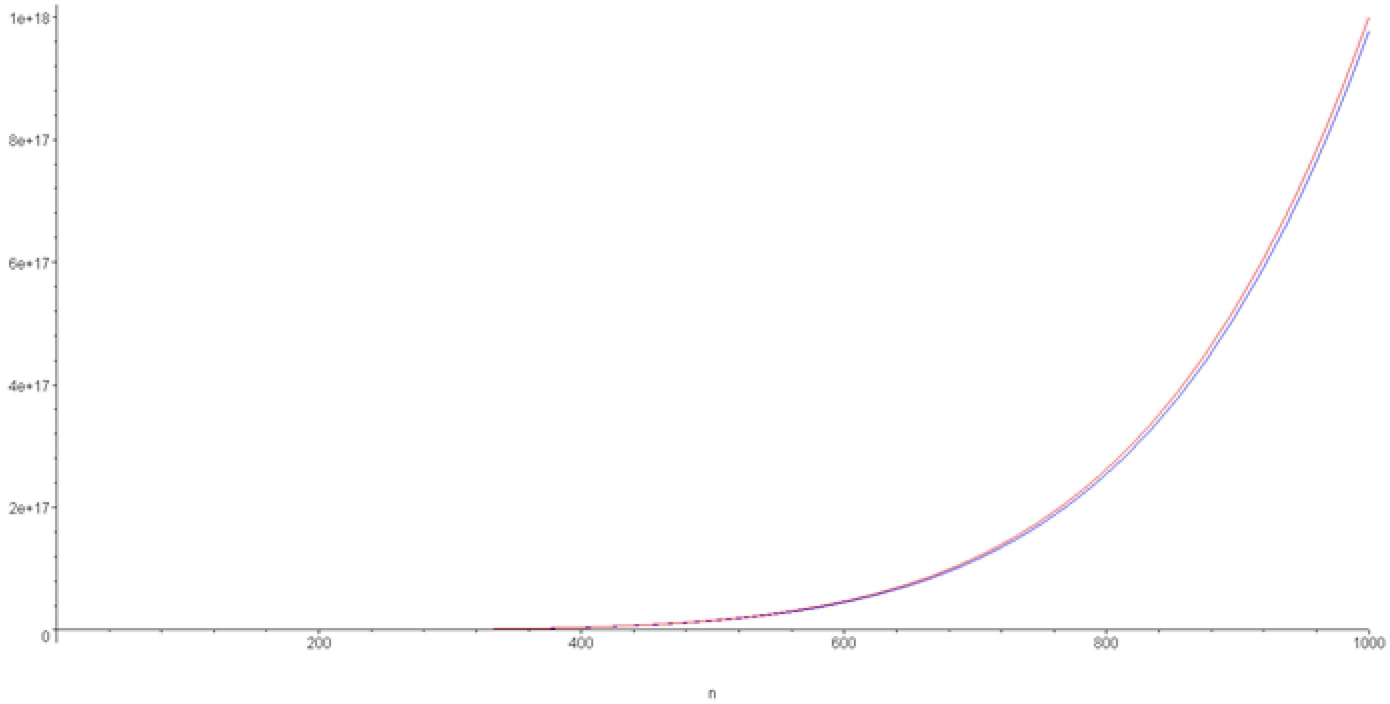
Polynomial Growth

- However, as we extend the range to [0, 100], they appear to look a lot more similar:



Polynomial Growth

- And finally, around *n* = 1000, the relative difference is less than 3%



Polynomial Growth

- **The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:**

n^2 **in the first case**

n^6 **in the second**

Polynomial Growth

- **Suppose however, that the coefficients of the leading terms were different**

Polynomial Growth

- **Suppose however, that the coefficients of the leading terms were different**

- **In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger**

Counting Instructions

Counting Instructions

- **Suppose we had two algorithms which sorted a list of size n and the number of instructions is given by**

$$b_{\text{worst}}(n) = 4.5n^2 - 0.5n + 5 \qquad \text{Sort B}$$

$$b_{\text{best}}(n) = 3.5n^2 + 0.5n + 5$$

$$s(n) = 4n^2 + 8n + 6 \qquad \text{Sort S}$$

Counting Instructions

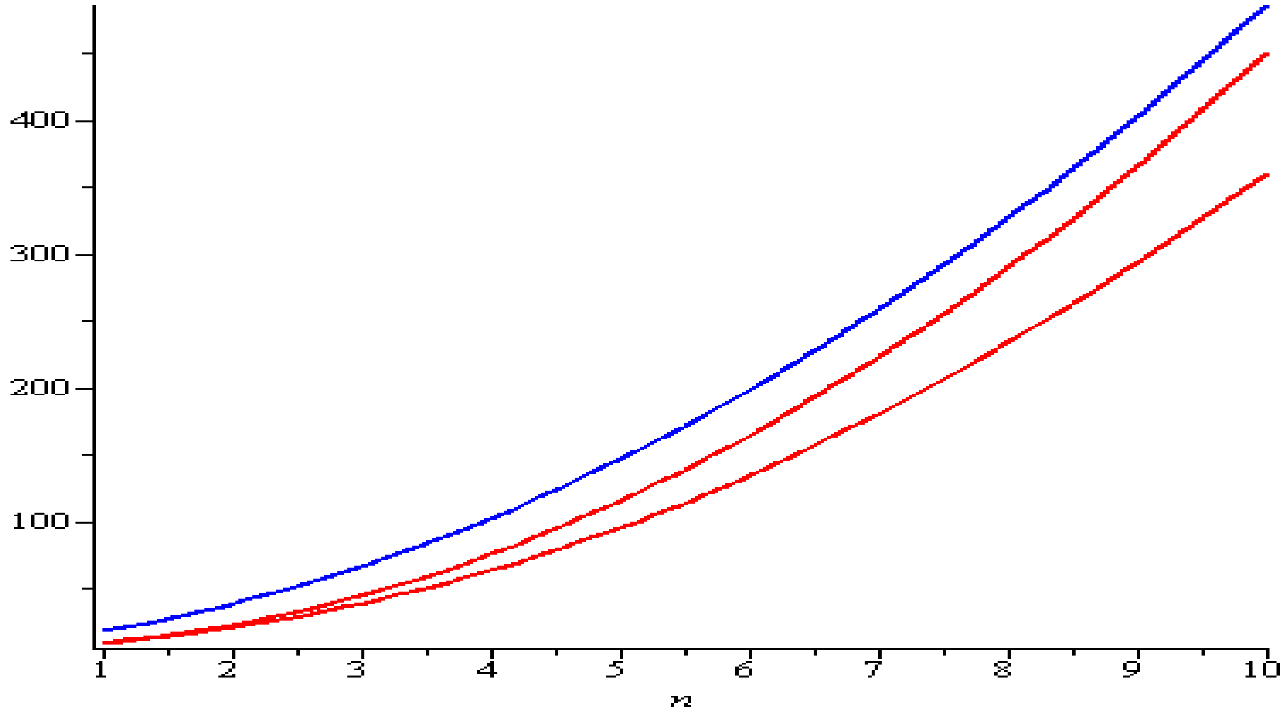
- **Suppose we had two algorithms which sorted a list of size *n* and the number of instructions is given by**

$$
\begin{array}{lll}
b_{worst}(n) & = 4.5n^2 - 0.5n + 5 & \text{Sort B} \\
b_{best}(n) & = 3.5n^2 + 0.5n + 5 & \\
s(n) & = 4n^2 + 8n + 6 & \text{Sort S}
\end{array}
$$

- **The smaller the value is, the fewer instructions are run**
  - For $n \leq 17$, $b_{worst}(n) < s(n)$
  - For $n \geq 18$, $b_{worst}(n) > s(n)$

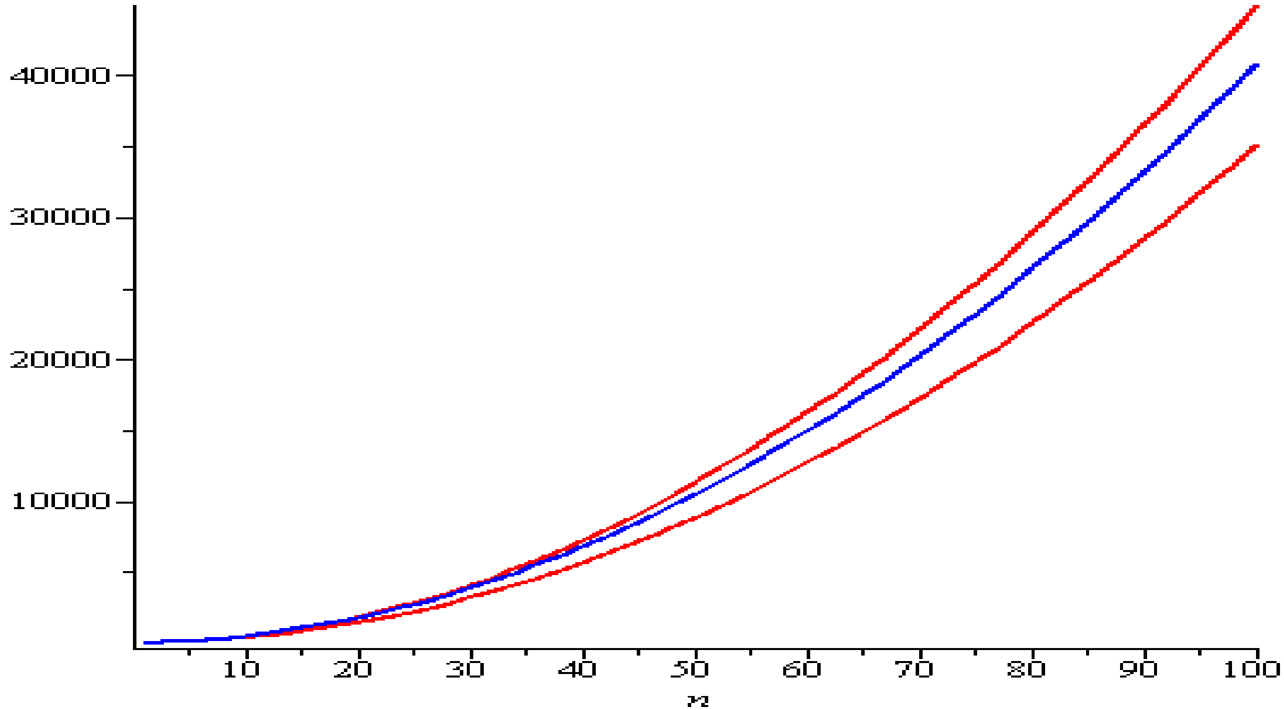
Counting Instructions

- **With small values of *n*, the algorithm described by s(*n*) requires more instructions than even the worst-case for sort b.**



Counting Instructions

- **With larger and larger lists, the number of instructions is essentially proportional to the leading coefficients**

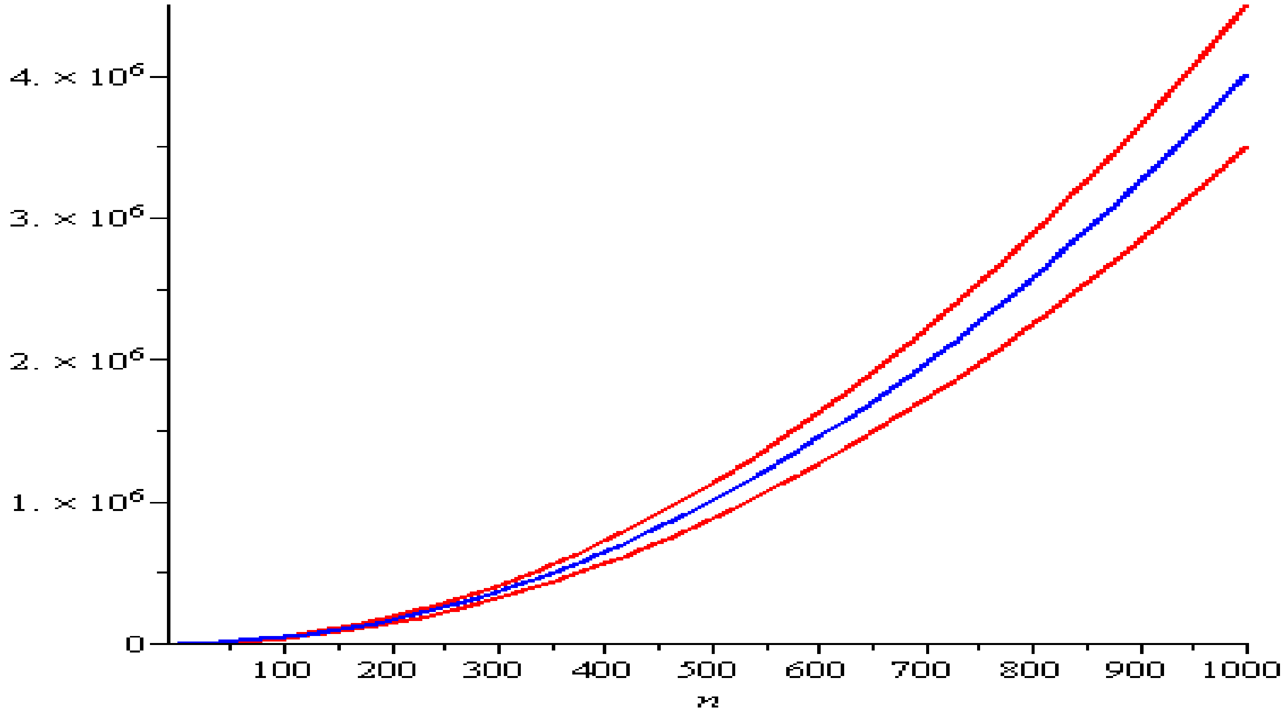


Counting Instructions

- **Near n = 1000,**

$$b_{\text{worst}}(n) \approx 1.125\, s(n)$$
$$b_{\text{best}}(n) \approx 0.875\, s(n)$$

- **Is this a significant difference?**



Asymptotic Analysis

- **Given an algorithm:**
  - **We need to be able to describe these values mathematically**
  - **We need a systematic means of using the description of the algorithm together with the properties of an associated data structure**
  - **We need to do this in a machine-independent way**

- **For this, we need Landau symbols and the associated asymptotic analysis**

Asymptotic Analysis

- **Big Idea**
  - **Ignore machine-dependent constants**
  - **Look at *growth* of *T(n)* as $n \to \infty$.**
- ***T*(*n*): the Asymptotic Running Time**
  - **Neglects the fact that the time cost of each statement actually depends on the compiler, interpreter and the hardware platform**
  - **Stands for the worst case**
  - ***T*(*n*) can be denoted or approximated by a function *f*(n)**

Landau Symbols

- **Before we begin, however, we will make some assumptions:**
  - **Our functions will describe the time or memory required to solve a problem of size *n***
  - **We conclude we are restricting ourselves to certain functions:**
    - **They are defined for $n \geq 0$ and $n \to \infty$**
    - **They are strictly positive for all *n***
      - **In fact, $f(n) > c$ for some value $c > 0$**
      - **That is, any problem requires at least one instruction and byte**
    - **They are non-decreasing (monotonic non-decreasing)**

Landau Symbols - Big Θ

- $f(n) = \Theta(g(n))$, if there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$ }
  - The function $f(n)$ has a rate of growth equal to that of $g(n)$

Landau Symbols - Big Θ

- These definitions are often unnecessarily tedious

- Note, however, that if f(n) and g(n) are polynomials of the same degree with positive leading coefficients such that:

$$\lim_{n\to\infty}\frac{\mathrm{f}(n)}{\mathrm{g}(n)} = c \quad \text{where} \quad 0 < c < \infty$$

Landau Symbols - Big Θ

- **Suppose that f(n) and g(n) satisfy** $\lim\limits_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = c$

- **From the definition, this means given $c > \varepsilon > 0$ there exists an $n_0 > 0$ such that** $\left|\frac{\mathrm{f}(n)}{\mathrm{g}(n)} - c\right| < \varepsilon$ **whenever** $n > n_0$

- **That is,** $c - \varepsilon < \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < c + \varepsilon$

$$\Rightarrow \mathrm{g}(n)(c-\varepsilon) < \mathrm{f}(n) < \mathrm{g}(n)(c+\varepsilon)$$

Landau Symbols - Big Θ

- Thus, the statement $g(n)\left(c-\varepsilon\right) < f(n) < g(n)\left(c+\varepsilon\right)$ says that f(n) = Θ(g(n))

- Note that this only goes one way:

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 < c < \infty$, it follows that

  f(n) = Θ(g(n))

Big Θ as an Equivalence Relation

- **Actually, $f(n) = \Theta(g(n))$ describes an equivalence relation:**
    1. **$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$**
    2. **$f(n) = \Theta(f(n))$**
    3. **If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, it follows that $f(n) = \Theta(h(n))$**

- **Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta Θ of each other**

Big Θ as an Equivalence Relation

- **For example, all of n^2**

$$100000\, n^2 - 4\, n + 19$$
$$n^2 + 1000000$$
$$323\, n^2 - 4\, n \ln(n) + 43\, n + 10$$
$$42n^2 + 32$$
$$n^2 + 61\, n \ln^2(n) + 7n + 14 \ln^3(n) + \ln(n)$$

are big- Θ of each other

E.g., $42n^2 + 32 = \Theta(\, 323\, n^2 - 4\, n \ln(n) + 43\, n + 10\,)$

Big Θ as an Equivalence Relation

- **For simple, we select one element to represent the class of these functions: n^2**
  - **We could chose any function, but this is the simplest**
  - **Drop low-order terms**
  - **Ignore leading constants.**

- **Example:** $\boxed{3}n^2 + \underbrace{90n - 5 \log n + 6046}_{\text{Drop}} = \Theta(n^2)$

Ignore **Drop**

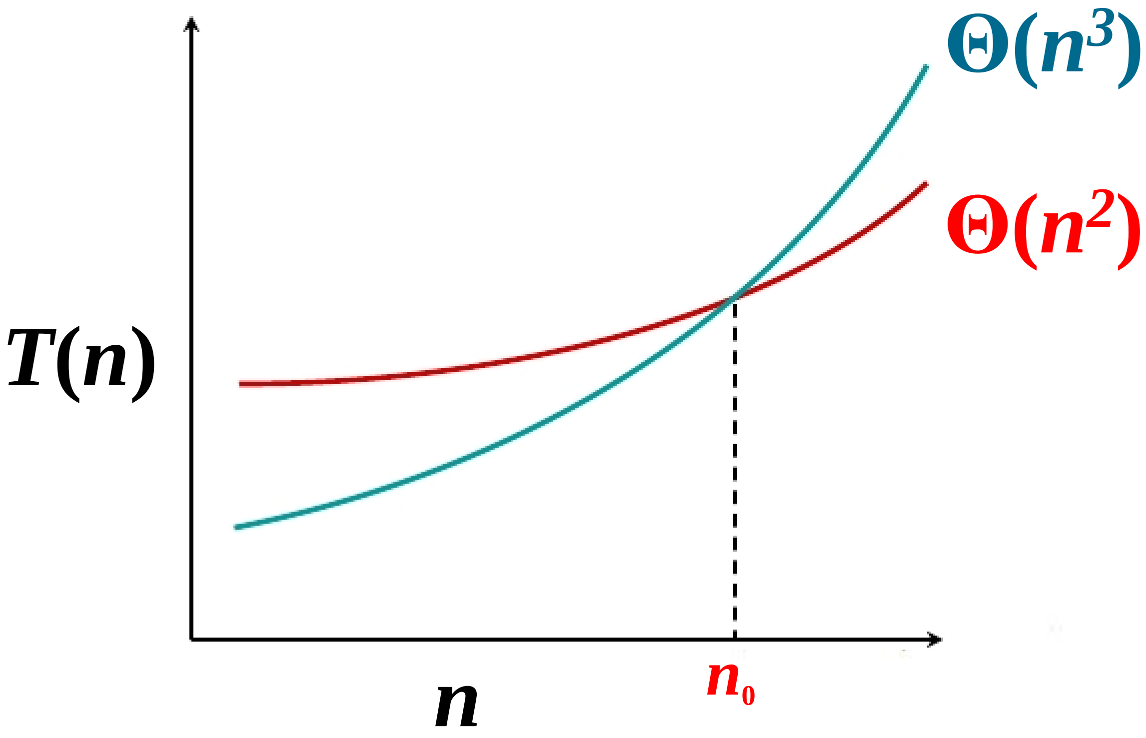
Big Θ as an Equivalence Relation

- **The most common classes are given names:**

$\Theta(1)$	constant
$\Theta(\log(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \log(n))$	"$n \log n$"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$\Theta(n!)$	factorial
$2^n, e^n, 4^n, \ldots$	exponential

Big Θ as an Equivalence Relation

- **When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.**



Big Θ as an Equivalence Relation

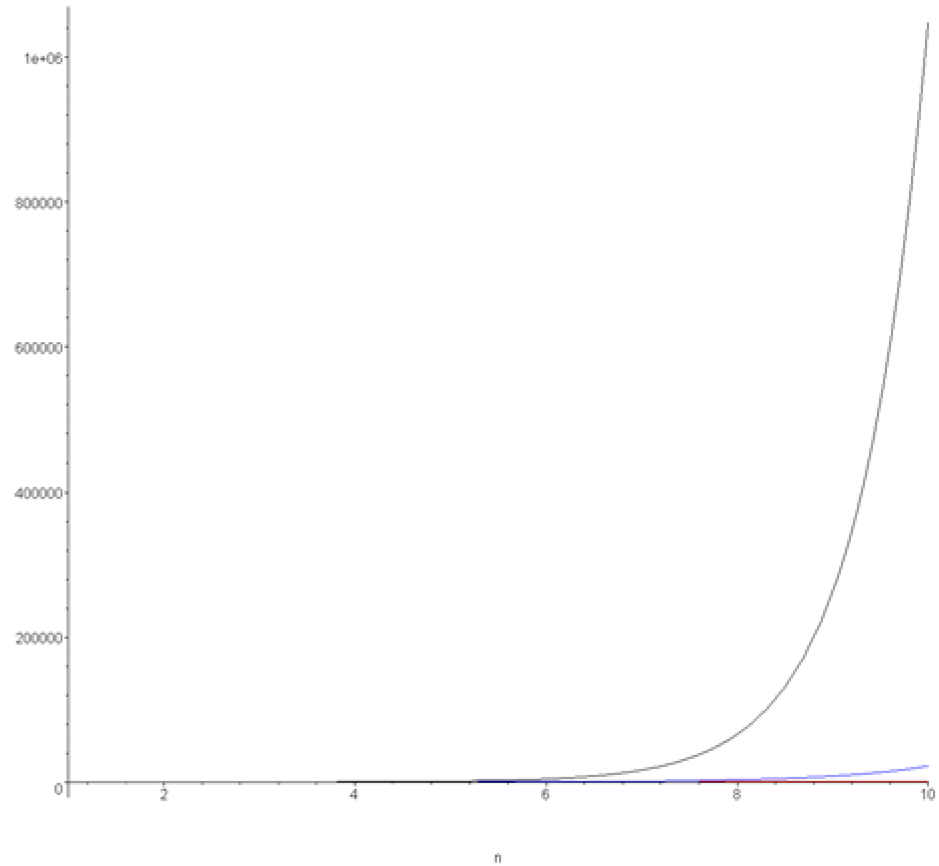
- **Recall that all logarithms are scalar multiples of each other:** $\boxed{\lim_{n\to\infty}\frac{\log_a n}{\ln n} = \lim_{n\to\infty}\left(\frac{\ln n/\ln a}{\ln n}\right) = \ln a}$
  - **Therefore $\log_b(n) = \Theta(\ln(n))$ for any base b**

- **Alternatively, there is no single equivalence class for exponential functions:**
  - **If $1 < a < b$,** $\boxed{\lim_{n\to\infty}\frac{a^n}{b^n} = \lim_{n\to\infty}\left(\frac{a}{b}\right)^n = 0}$

- **However, we will see that it is almost universally unacceptable to have an exponentially growing function!**

Big Θ as an Equivalence Relation

- Plotting 2^n, e^n, and 4^n on the range [1, 10] already shows how significantly different the functions grow



Note:

$2^{10} = 1{,}024$

$e^{10} \approx 22{,}026$

$4^{10} = 1{,}048{,}576$

Landau Symbols - Big O

- $f(n) = O(g(n))$, if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ }

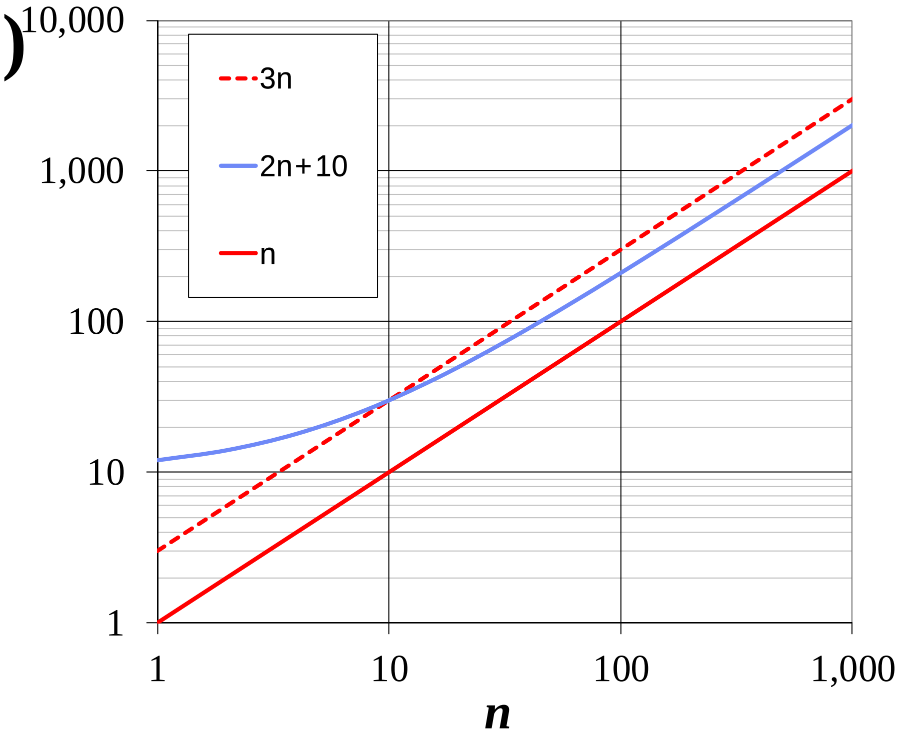
- Similar to big Θ, we have another definition that

  If $\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$ where $0 < c < \infty$, it follows that

  $f(n) = O(g(n))$

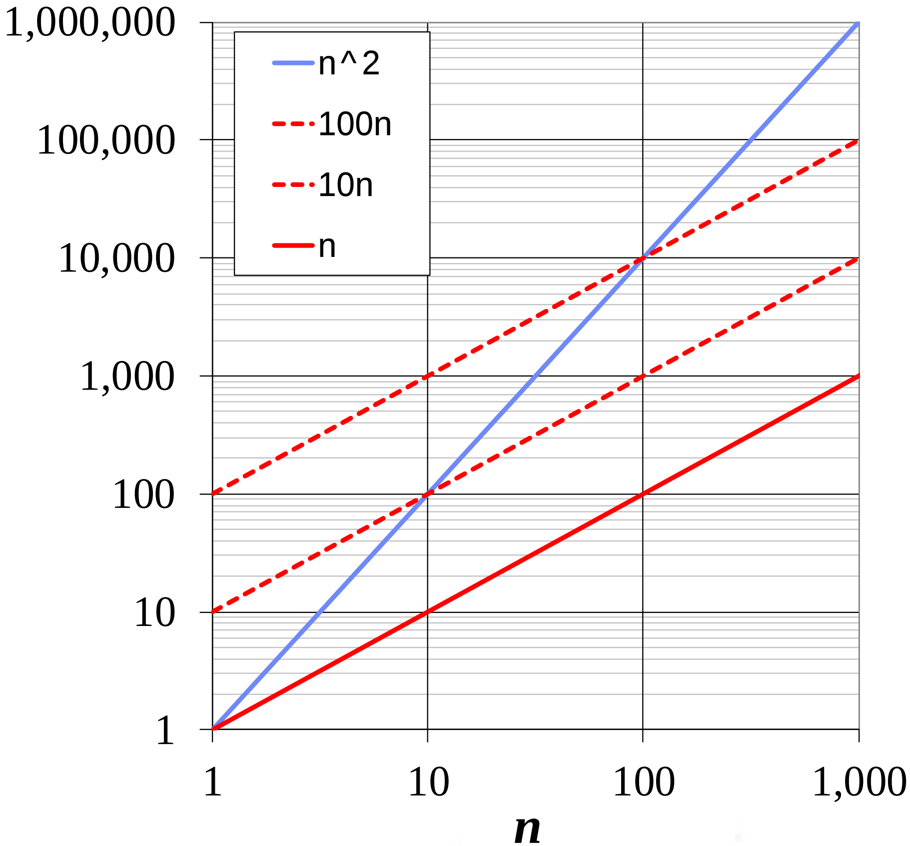
Landau Symbols - Big O

- **Example: $2n + 10$ is $O(n)$**
  - **Proof:**
  - **$2n + 10 \leq cn$**
    $\Rightarrow (c - 2)\, n \geq 10$
    $\Rightarrow n \geq 10/(c - 2)$
    Pick $c = 3$ and $n_0 = 10$



Landau Symbols - Big O

- **Example: n^2 is not O(n)**
  - **Proof:**
    $n^2 \leq c \cdot n$
    $\Rightarrow n \leq c$
    The above inequality cannot be satisfied since *c* must be a constant while $n \rightarrow \infty$



Landau Symbols - Big O

- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - need $c > 0$ and $n_0 \geq 1$ such that
    $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
    => true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$ is $O(\log n)$
  - need $c > 0$ and $n_0 \geq 1$ such that
    $3 \log n + 5 \leq c \log n$ for $n \geq n_0$
    => true for $c = 8$ and $n_0 = 2$

General Rules for calculating Big O

- ***Ignore leading constants.***
  - For $d > 0$, $O(f(n)) = O(d \cdot f(n))$.

    Proof:

    $O(f(n)) = g(n) \Rightarrow g(n) \leq c \cdot f(n)$ for all $n \geq n_0$

    $\Rightarrow g(n) \leq (c/d) \cdot (d \cdot f(n))$ for all $n \geq n_0$

  - **Thus, we practically prefer saying that the time complexity of program A is $O(n)$ rather than $O(6n)$.**

General Rules for calculating Big O

- ***Drop** low-order terms.*
  - $2^n + n^3$ is $O(2^n)$.

    Proof:
    need $c > 0$ and $n_0 \geq 1$ such that
    $2^n + n^3 \leq c \cdot 2^n$ for $n \geq n_0$
    => true for $c = 2$ and $n_0 = 10$
    => actually, we can drop n^3 at the beginning

    since $$\lim_{n \to \infty} \frac{n^3}{2^n} = 0$$

  - **Every exponential grows faster than a polynomial.**

Tightness of Big O

- $3n^3 + 20n^2 + 5$ is $O(n^3)$ ⇨ Tight bound, $= \Theta(n^3)$
- Naturally, we can prove that $3n^3 + 20n^2 + 5$ is $O(n^4)$ ⇨ Not a tight bound
  - We generally prefer a tight bound on big O (or else the big Θ), when we can prove it.
  - However, an upper bound is acceptable under many circumstances

Big-Oh Operations

- $O(f(n))+O(g(n)) = O(\max\{f(\mathrm{n}), g(n)\})$
- $O(f(n))+O(g(n)) = O(f(\mathrm{n})+g(n))$
- $O(f(n)) \cdot O(g(n)) = O(f(\mathrm{n}) \cdot g(n))$
- $O(c \cdot f(n)) = O(f(\mathrm{n}))$

Important Functions

- **The most common classes are given names:**

O (1)	**constant**
O (log(n))	**logarithmic**
O (n)	**linear**
O (n log(n))	**"*n* log *n*"**
O (n^2)	**quadratic**
O (n^3)	**cubic**
O (n!)	**factorial**
2^n, e^n, 4^n, ...	**exponential**

O(n)

- $h = 0$

  ```
  for (i = 0; i < n; i++)
  {
      h += i;
  }
  ```

O(n^2)

- ```
  h = 0;
  for (i = 0; i < n; i++)
  {
        for (j = 0; j < n; j++)
        {
              h += i * j;
        }
  }
  ```
  ```

O(n^2)

- $h = 0;$
- for ($i = 0$; $i < n$; i++)
  {

  for ($j = 0$; $j < i$; j++)

  {

  h += i * j;

  }

  }

- **Since the second loop depends on the first, we can denote the looping step number by**

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i} 1 or 2 or 3 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

which is $O(n^2)$.

O(log *n*)

- h=0, k = 1;
- while(k<=n)

  {

  h += k;

  k = 2*k;

  }
- In this case, the index k jumps (i.e. k takes on values {1, 2, 4 …}) till n is exceeded.
- There will be log (n) + 1 steps, therefore the complexity is O(log n).

Landau Symbols - Big O

- $f(n) = O(g(n))$, if there exist positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$ }

- Similar to big Θ, we have another definition that

If $\boxed{\lim_{n \to \infty} \frac{f(n)}{g(n)} \le c}$ where $\boxed{0 < c < \infty}$, it follows that

$f(n) = O(g(n))$

Landau Symbols - Big Ω

- $f(n) = \Omega(g(n))$, if there exist positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0$ }

- Similar to big Θ, we have another definition that

If $\boxed{\lim_{n\to\infty} \frac{f(n)}{g(n)} \ge c}$ where $\boxed{0 < c < \infty}$, it follows that

$f(n) = \Omega(g(n))$

The Relationship between Θ, O and Ω

- **O - Upper bounds.**
  - **f(n) is O(g(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have f(n) $\leq c \cdot$ g(n).**

- **Ω - Lower bounds.**
  - **f(n) is Ω(g(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n0$ we have f(n) $\geq c \cdot$ g(n).**

- **Θ - Tight bounds.**
  - **f(n) is Θ(g(n)) if f(n) is both O(g(n)) and Ω(g(n)).**

Intuition for Θ, O and Ω

- **O**
  - **f(n) is O(g(*n*)) if f(*n*) is asymptotically less than or equal to g(*n*)**
- **Ω**
  - **f(*n*) is Ω(g(*n*)) if f(*n*) is asymptotically greater than or equal to g(*n*)**
- **Θ**
  - **f(*n*) is Θ(g(*n*)) if f(*n*) is asymptotically equal to g(*n*)**

Another Intuition for Θ, O and Ω

- **O**
  - $f(n) = O(g(n)) \approx a \leq b$

    $f(n)$ $\quad$ $c \cdot g(n)$
- **Ω**
  - $f(n) = \Omega(g(n)) \approx a \geq b$
- **Θ**
  - $f(n) = \Theta(g(n)) \approx a = b$

Example for Θ, O and Ω

- $f(n) = 32n^2 + 17n + 32$
  - $f(n)$ is $O(n^2)$, $O(n^3)$
  - $f(n)$ is $\Omega(n^2)$, $\Omega(n)$
  - $f(n)$ is $\Theta(n^2)$

  - $f(n)$ is not $O(n)$
  - $f(n)$ is not $\Omega(n^3)$
  - $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

Five Landau Symbols

- **We sometimes use these five possible descriptions:**

$$f(n) = o(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = O(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \Theta(g(n)) \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \Omega(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) = \omega(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Five Landau Symbols

- **Graphically, we can summarize these as follows:**

We say $f(n) =$ $\mathbf{O}(g(n))$ $\Omega(g(n))$

$\mathbf{o}(g(n))$ $\Theta(g(n))$ $\omega(g(n))$

if $\lim_{n \to \infty} \frac{f(n)}{g(n)} =$ 0 $\quad 0 < c < \infty \quad$ ∞

- **In the following , we intend to use big O and big Θ mostly.**

Outline

In this topic, we will examine code to determine the run time of various operations

We will calculate the run times of:

- Operators +, -, =, +=, ++, etc.
- Control statements `if`, `for`, `while`, `do-while`, `switch`
- Functions
- Recursive functions

3.2 ANALYSIS of OPERATIONS

Motivation

The goal of algorithm analysis is to take a block of code and determine the asymptotic run time or asymptotic memory requirements based on various parameters

- **Given an array of size n:**
  - **Selection sort requires $\Theta(n^2)$ time**
  - **Merge sort, quick sort, and heap sort all require $\Theta(n \log(n))$ time**

- **However:**
  - **Merge sort requires $\Theta(n)$ additional memory**
  - **Quick sort requires $\Theta(\log(n))$ additional memory**
  - **Heap sort requires $\Theta(1)$ memory**

Motivation

The asymptotic behaviors of algorithms indicates the ability to scale

– Suppose we are sorting an array of size n

Selection sort has a run time of $\Theta(n^2)$

– $2n$ entries requires $(2n)^2 = 4n^2$

- Four times as long to sort

– $10n$ entries requires $(10n)^2 = 100n^2$

- One hundred times as long to sort

Motivation

The other sorting algorithms have $\Theta(n \log(n))$ run times

- $2n$ entries require $(2n) \log(2n)$
  $= (2n) (\log(n) + 1)$
  $= 2(n \log(n)) + 2n$

- $10n$ entries require $(10n) \log(10n)$
  $= (10n) (\log(n) + \log(10))$
  $= 10(n \log(n)) + 10n*\log(10)$

In each case, it requires $\Theta(n)$ more time

However:

- Merge sort will require twice and 10 times as much memory
- Quick sort will require one or four additional memory locations
- Heap sort will not require any additional memory

Motivation

If we are storing objects which are not related, the hash table has, in many cases, optimal characteristics:

- Many operations are $\Theta(1)$
- *I.e.*, the run times are independent of the number of objects being stored

If we are required to store both objects and relations, both memory and time will increase

- Our goal will be to minimize this increase

Motivation

To properly investigate the determination of run times asymptotically, we will discuss

– Operations

– Control statements

  • Conditional statements and loops

– Functions

– Recursive functions

Operators

Because each machine instruction can be executed in a fixed time, we may assume each operation requires a fixed time

- **The time required for any operator is $\Theta(1)$ including:**
  - **Retrieving/storing variables from memory**
  - **Variable assignment** `=`
  - **Integer operations** `+ - * / % ++ --`
  - **Logical operations** `&& || !`
  - **Bitwise operations** `& | ^ ~`
  - **Relational operations** `== != < <= => >`
  - **Memory allocation and deallocation** `new delete`

Operators

Of these, memory allocation and deallocation are the slowest by a significant factor

- A quick test on `unix` shows a factor of over 100
- They require communication with the operation system
- This does not account for the time required to call the constructor and destructor

Note that after memory is allocated, the constructor is run

- The constructor may not run in $\Theta(1)$ time

Blocks of Operations

Each operation runs in $\Theta(1)$ time and therefore any fixed number of operations also run in $\Theta(1)$ time, for example:

– Swap variables a and b

```
int tmp = a;
a = b;
b = tmp;
```

– Update a sequence of values

```
++index;
prev_modulus = modulus;
modulus = next_modulus;
next_modulus = modulus_table[index];
```

Blocks in Sequence

Suppose you have now analyzed a number of blocks of code run in sequence

```
template <typename T>
void update_capacity( int delta ) {
   T *array_old = array;
   int capacity_old = array_capacity;
   array_capacity += delta;
   array = new T[array_capacity];

   for ( int i = 0; i < capacity_old; ++i ) {
         array[i] = array_old[i];
   }

   delete[] array_old;
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

To calculate the total run time, add the entries: $\Theta(1 + n + 1) = \mathbf{\Theta(n)}$

Blocks in Sequence

Other examples include:

- Run three blocks of code which are $\Theta(1)$, $\Theta(n^2)$, and $\Theta(n)$
  total run time $\Theta(1 + n^2 + n) = \Theta(n^2)$

- Run two blocks of code which are $\Theta(n \log(n))$, and $\Theta(n^{1.5})$
  total run time $\Theta(n \log(n) + n^{1.5}) = \Theta(n^{1.5})$

When considering a sum, take the **dominant term**

Control Statements

Next we will look at the following control statements

These are statements which potentially alter the execution of instructions

- Conditional statements
  `if`, `switch`

- Condition-controlled loops
  `for`, `while`, `do-while`

- Count-controlled loops
  `for i from 1 to 10 do ... end do;`

- Collection-controlled loops
  `foreach ( int i in array ) { ... }`

Control Statements

Given any collection of nested control statements, it is always necessary to work inside out

- **Determine the run times of the inner-most statements and work your way out**

Control Statements

Given

```
if ( condition ) {
    // true body
} else {
    // false body
}
```

The run time of a conditional statement is:

- the run time of the condition (the test), plus
- the run time of the body which is run

In most cases, the run time of the condition is $\Theta(1)$

Control Statements

In some cases, it is easy to determine which statement must be run:

```
int factorial ( int n ) {
        if ( n == 0 ) {
                return 1;
        } else {
                return n * factorial ( n – 1 );
        }
}
```

Control Statements

- In others, it is less obvious
  - Find the maximum entry in an array:

```
int find_max( int *array, int n ) {
    max = array[0];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

Analysis of Statements

In this case, we don’t know

If we had information about the distribution of the entries of the array, we may be able to determine it

– if the list is sorted (ascending) it will always be run
– if the list is sorted (descending) it will be run once
– if the list is uniformly randomly distributed, then???

Condition-controlled Loops

The C++ for loop is a condition controlled statement:

```
for ( int i = 0; i < N; ++i ) {
        // ...
}
```

is identical to

```
int i = 0;                              // initialization
while ( i < N ) {                       // condition
    // ...
    ++i;                                // increment
}
```

Condition-controlled Loops

The initialization, condition, and increment usually are single statements running in $\Theta(1)$

```
for ( int i = 0; i < N; ++i ) {
        // ...
}
```

Condition-controlled Loops

The initialization, condition, and increment statements are usually $\Theta(1)$

For example,

```
for ( int i = 0; i < n; ++i ) {
    // ...
}
```

Thus, the run time is at $\Omega(1)$, that is, at least the initialization and one condition must occur

Condition-controlled Loops

If the body does not depend on the variable (in this example, *i*), then the run time of

```
for ( int i = 0; i < n; ++i ) {
    // code which is Theta(f(n))
}
```

is $\Theta(n \, \mathrm{f}(n))$

If the body is $\mathrm{O}(\mathrm{f}(n))$, then the run time of the loop is $\mathrm{O}(n \, \mathrm{f}(n))$

Condition-controlled Loops

For example,

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

This code has run time

$$\Theta(n \cdot 1) = \Theta(n)$$

Condition-controlled Loops

Another example example,

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

The previous example showed that the inner loop is $\Theta(n)$, thus the outer loop is

$$\Theta(n \cdot n) = \Theta(n^2)$$

Conditional Statements

Consider this example

```
void Disjoint_sets::clear() {
    if ( sets == n ) {
        return;
    }

    max_height = 0;
    num_disjoint_sets = n;

    for ( int i = 0; i < n; ++i ) {
        parent[i] = i;
        tree_height[i] = 0;
    }
}
```

$\Theta(1)$

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

$$\mathrm{T}_{\mathrm{clear}}(n) = \begin{cases} \Theta(1) & sets = n \\ \Theta(n) & \text{otherwise} \end{cases}$$

Analysis of Repetition Statements

Suppose with each loop, we use a linear search an array of size m:

```
for ( int i = 0; i < n; ++i ) {
    // search through an array of size m
       Execution Body
    // O( m );
}
```

The inner loop is $O(m)$ and thus the outer loop is

$$O(n \cdot m)$$

Analysis of Repetition Statements

If the body does depends on the variable (in this example, *i*), then the run time of

```
for ( int i = 0; i < n; ++i ) {
    // code which is Theta(f(i,n))
}
```

is $\Theta\left(1+\sum_{i=0}^{n-1} 1+\mathrm{f}(i,n)\right)$**; and if the body is** $\mathrm{O}(\mathrm{f}(i, n))$**, then the result is** $\mathrm{O}\left(1+\sum_{i=0}^{n-1} 1+\mathrm{f}(i,n)\right)$

Analysis of Repetition Statements

For example,

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

The inner loop is $\mathrm{O}(1 + i(1 + 1)) = \Theta(i)$ hence

the outer is $\Theta\left(1+\sum_{i=0}^{n-1} 1+i\right) = \Theta\left(1+n+\sum_{i=0}^{n-1} i\right)$

$$= \Theta\left(1+n+\frac{n(n-1)}{2}\right) = \Theta\left(n^2\right)$$

Analysis of Repetition Statements

As another example:

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        for ( int k = 0; k < j; ++k ) {
            sum += i + j + k;
        }
    }
}
```

From inside to out:

$\Theta(1)$

$\Theta(j)$

$\Theta(i^2)$

$\Theta(n^3)$

Control Statements

Switch statements appear to be nested if statements:

```
switch( i ) {
    case 1:   /* do stuff */ break;
    case 2:   /* do other stuff */ break;
    case 3:   /* do even more stuff */ break;
    case 4:   /* well, do stuff */ break;
    case 5:   /* tired yet? */ break;
    default:  /* do default stuff */
}
```

Control Statements

Thus, a switch statement would appear to run in $O(n)$ time where n is the number of cases, the same as nested if statements

– Then why not use:

```
if ( i == 1 ) { /* do stuff */ }
else if ( i == 2 ) { /* do other stuff */ }
else if ( i == 3 ) { /* do even more stuff */ }
else if ( i == 4 ) { /* well, do stuff */ }
else if ( i == 5 ) { /* tired yet? */ }
else { /* do default stuff */ }
```

Control Statements

However, switch statements were included in the original C language... why?

First, you may recall that the cases must be actual values, either:

- integers
- characters

For example, you cannot have a case with a variable, e.g.,

```
case n: /* do something */ break;  //bad
```

Control Statements

The compiler looks at the different cases and calculates an appropriate jump

For example, assume:

- the cases are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- each case requires a maximum of 24 bytes (for example, six instructions)

Then the compiler simply makes a jump size based on the variable, jumping ahead either 0, 24, 48, 72, ..., or 240 instructions

Serial Statements

Suppose we run one block of code followed by another block of code

Such code is said to be run *serially*

If the first block of code is $\mathrm{O}(\mathrm{f}(n))$ and the second is $\mathrm{O}(\mathrm{g}(n))$, then the run time of both two blocks is

$$\mathrm{O}(\,\mathrm{f}(n) + \mathrm{g}(n)\,)$$

which usually (for algorithms not including function calls) simplifies to one or the other

Serial Statements

Consider the following two problems:

- search through a random list of size n to find the maximum entry, and
- search through a random list of size n to find if it contains a particular entry

What is the proper means of describing the run time of these two algorithms?

Serial Statements

Searching for the maximum entry requires that each element in the array be examined

- thus, it must run in $\Theta(n)$ time

Searching for a particular entry may end earlier

- for example, the first entry we are searching for may be the one we are looking for, thus, it runs in $O(n)$ time

Serial Statements

Therefore,

- if the leading term is big-Θ, then the result must be big-Θ, otherwise
- if the leading term is big-O, we can say the result is big-O

For example,

$$\mathrm{O}(n) + \mathrm{O}(n^2) + \mathrm{O}(n^4) = \mathrm{O}(n + n^2 + n^4) = \mathrm{O}(n^4)$$

$$\mathrm{O}(n) + \Theta(n^2) = \Theta(n^2)$$

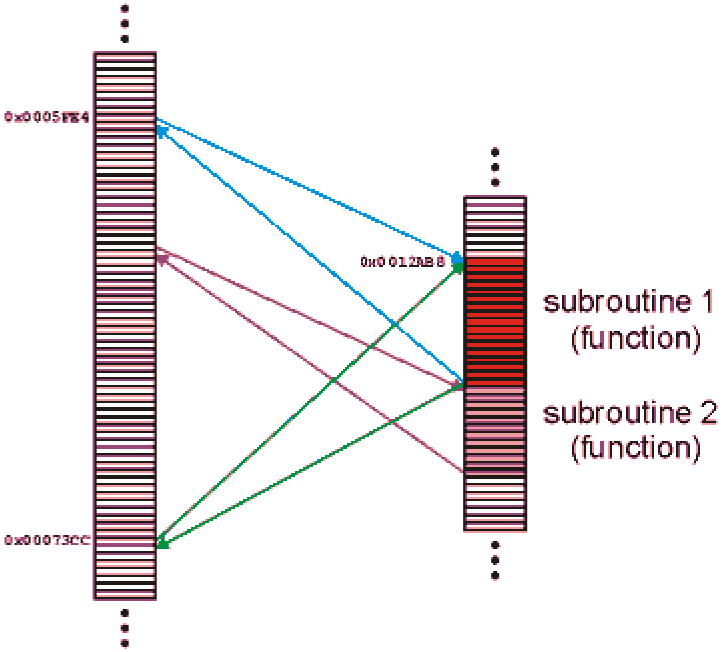
$$\mathrm{O}(n^2) + \Theta(n) = \mathrm{O}(n^2)$$

$$\mathrm{O}(n^2) + \Theta(n^2) = \Theta(n^2)$$

Functions

A function (or subroutine) is code which has been separated out, either to:

- **and repeated operations**
  - **e.g., mathematical functions**
- **group related tasks**
  - **e.g., initialization**



Functions

Because a subroutine (function) can be called from anywhere, we must:

- **prepare the appropriate environment**
- **deal with arguments (parameters)**
- **jump to the subroutine**
- **execute the subroutine**
- **deal with the return value**
- **clean up**

Functions

Fortunately, this is such a common task that all modern processors have instructions that perform most of these steps in one instruction

Thus, we can assume that the overhead required to make a function call and to return is $\Theta(1)$

Functions

Because any function requires the overhead of a function call and return, we will always assume that

$$T_f = \Omega(1)$$

That is, it is impossible for any function call to have a zero run time

Functions

Thus, given a function $\mathrm{f}(n)$ (the run time of which depends on n) we will associate the run time of $\mathrm{f}(n)$ by some function $\mathrm{T}_{\mathrm{f}}(n)$

– We may write this to $\mathrm{T}(n)$

Because the run time of any function is at least $\mathrm{O}(1)$, we will include the time required to both call and return from the function in the run time

Functions

Consider this function:

```
void Disjoint_sets::set_union( int m, int n ) {

        m = find( m );
        n = find( n );

        if ( m == n ) {
                return;
        }

        --num_disjoint_sets;

        if ( tree_height[m] >= tree_height[n] ) {
            parent[n] = m;

            if ( tree_height[m] == tree_height[n] ) {
                ++( tree_height[m] );
                max_height = std::max( max_height, tree_height[m] );
            }
        } else {
            parent[m] = n;
        }
}
```

$2T_{\text{find}}$

$\Theta(1)$

$$T_{\text{set\_union}} = 2T_{\text{find}} + \Theta(1)$$

Recursive Functions

A function is relatively simple (and boring) if it simply performs operations and calls other functions

Most interesting functions designed to solve problems usually end up calling themselves

- **Such a function is said to be *recursive***

Recursive Functions

As an example, we could implement the factorial function recursively:

```
int factorial( int n ) {
    if ( n <= 1 ) {
        return 1;
    } else {
        return n * factorial( n – 1 );
    }
}
```

$\Theta(1)$

$T_!(n-1)+\Theta(1)$

Recursive Functions

Thus, we may analyze the run time of this function as follows:

$$\mathrm{T}_!(n) = \begin{cases} \Theta(1) & n \le 1 \\ \mathrm{T}_!(n-1) + \Theta(1) & n > 1 \end{cases}$$

We don't have to worry about the time of the conditional ($\Theta(1)$) nor is there a probability involved with the conditional statement

Recursive Functions

The analysis of the run time of this function yields a recurrence relation:

$$T_!(n) = T_!(n - 1) + \Theta(1) \qquad T_!(1) = \Theta(1)$$

This recurrence relation has Landau symbols and we replace each Landau symbol with a representative function:

$$T_!(n) = T_!(n - 1) + 1 \qquad T_!(1) = 1$$

Recursive Functions

We can examine the first few steps:

$$
\begin{aligned}
T_!(n) &= T_!(n-1) + 1 \\
&= T_!(n-2) + 1 + 1 = T_!(n-2) + 2 \\
&= T_!(n-3) + 3
\end{aligned}
$$

Recursive Functions

We can examine the first few steps:

$$\begin{aligned} T_!(n) &= T_!(n-1) + 1 \\ &= T_!(n-2) + 1 + 1 = T_!(n-2) + 2 \\ &= T_!(n-3) + 3 \end{aligned}$$

Recursive Functions

We can examine the first few steps:

$$\begin{aligned} T_!(n) &= T_!(n-1) + 1 \\ &= T_!(n-2) + 1 + 1 = T_!(n-2) + 2 \\ &= T_!(n-3) + 3 \end{aligned}$$

From this, we see a pattern:

$$T_!(n) = T_!(n-k) + k$$

When $k = n - 1$, …

$$T_!(n) = T_!(n-k) + k$$

Recursive Functions

If $k = n - 1$ then

$$
\begin{aligned}
T_{!}(n) &= T_{!}(n - (n - 1)) + n - 1 \\
&= T_{!}(1) + n - 1 \\
&= 1 + n - 1 = n
\end{aligned}
$$

Recursive Functions

If $k = n - 1$ then

$$
\begin{aligned}
T_!(n) &= T_!(n - (n - 1)) + n - 1 \\
&= T_!(1) + n - 1 \\
&= 1 + n - 1 = n
\end{aligned}
$$

Recursive Functions

If $k = n - 1$ then

$$
\begin{aligned}
T_!(n) &= T_!(n - (n - 1)) + n - 1 \\
&= T_!(1) + n - 1 \\
&= 1 + n - 1 = n
\end{aligned}
$$

Thus, $T_!(n) = \Theta(n)$

Recursive Functions

Suppose we want to sort a array of n items

Recursive Functions

Suppose we want to sort a array of *n* items

We could:

- **go through the list and find the largest item**
- **swap the last entry in the list with that largest item**
- **then, go on and sort the rest of the array**

Recursive Functions

Suppose we want to sort a array of n items

We could:

- **go through the list and find the largest item**
- **swap the last entry in the list with that largest item**
- **then, go on and sort the rest of the array**

This is called *selection sort*

Recursive Functions

```
void sort( int * array, int n ) {
    if ( n <= 1 ) {
        return;                         // special case:  0 or 1 items are always sorted
    }

    int posn = 0;                       // assume the first entry is the smallest
    int max = array[posn];

    for ( int i = 1; i < n; ++i ) {     // search through the remaining entries
        if ( array[i] > max ) {         // if a larger one is found
            posn = i;                   // update both the position and value
            max = array[posn];
        }
    }

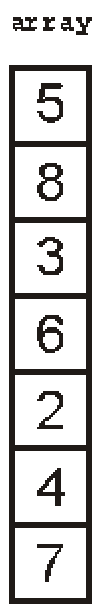
    int tmp = array[n - 1];             // swap the largest entry with the last
    array[n - 1] = array[posn];
    array[posn] = tmp;

    sort( array, n – 1 );               // sort everything else
}
```

Recursive Functions

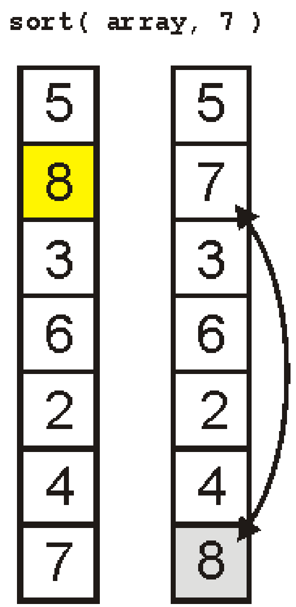
We could call this function as follows:

```
int *array = {5, 8, 3, 6, 2, 4, 7};
sort( array, 7 );   // sort an array of seven items
```



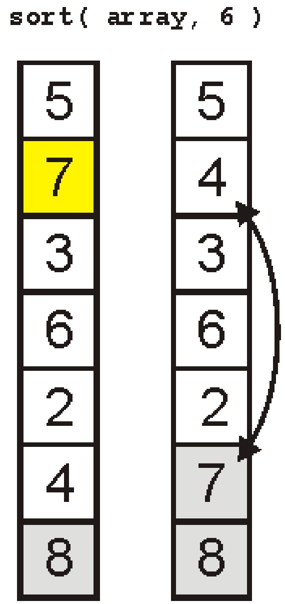
Recursive Functions

The first call finds the largest element



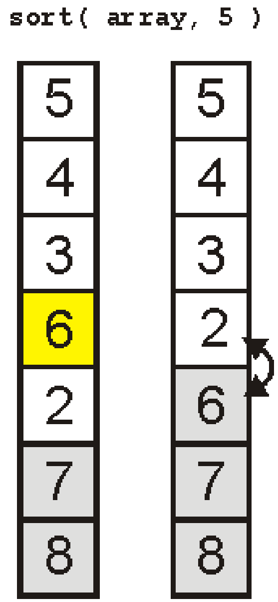
Recursive Functions

The next call finds the 2nd-largest element



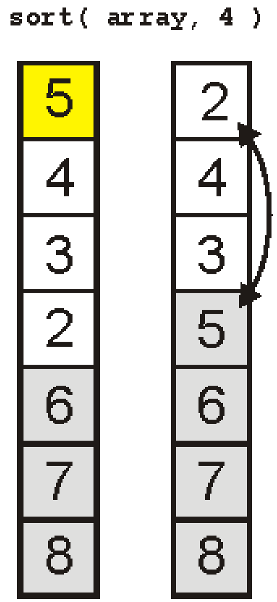
Recursive Functions

The third finds the 3rd-largest



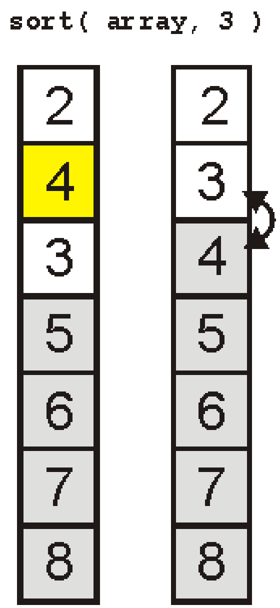
Recursive Functions

And the 4^{th}



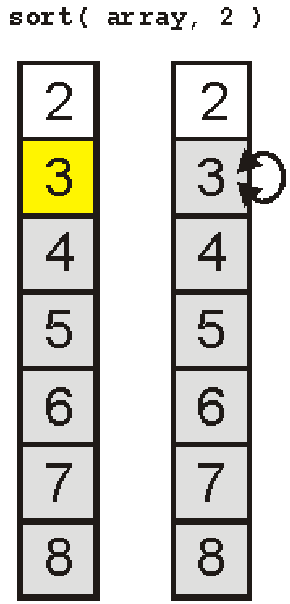
Recursive Functions

And the 5th



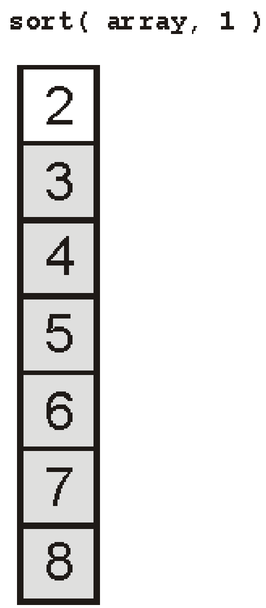
Recursive Functions

Finally the 6th



Recursive Functions

And the array is sorted:



Recursive Functions

Analyzing the function, we get:

```
void sort( int * array, int n ) {
    if ( n <= 1 ) {
        return;
    }

    int posn = 0;
    int max = array[posn];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            posn = i;
            max = array[posn];
        }
    }

    int tmp = array[n - 1];
    array[n - 1] = array[posn];
    array[posn] = tmp;

    sort( array, n - 1 );
}
```

$\mathrm{T}(0) = \mathrm{T}(1) = \Theta(1)$

$\Theta(1)$

$\Theta(1)$ $\Theta(n)$

$\Theta(1)$

$\mathrm{T}(n-1)$

$$\begin{aligned}\mathrm{T}(n) &= \Theta(1) + \Theta(n) + \Theta(1) + \mathrm{T}(n-1)\\ &= \mathrm{T}(n-1) + \Theta(n)\end{aligned}$$

Recursive Functions

Thus, replacing each Landau symbol with a representative, we are required to solve the recurrence relation:

$$\mathrm{T}(n) = \mathrm{T}(n - 1) + n \qquad \mathrm{T}(1) = 1$$

Recursive Functions

Consequently, the sorting routine has the run time

$$\mathrm{T}(n) = \Theta(n^2)$$

To see this by hand, consider the following

$$\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(n-1) + n \\
&= \big(\mathrm{T}(n-2) + (n-1)\big) + n \\
&= \mathrm{T}(n-2) + n + (n-1) \\
&= \mathrm{T}(n-3) + n + (n-1) + (n-2) \\
&\;\vdots \\
&= \mathrm{T}(1) + \sum_{i=2}^{n} i = 1 + \sum_{i=2}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\end{aligned}$$

Recursive Functions

Consider, instead, a binary search of a sorted list:

- check the middle entry
- if we do not find it, check either the left- or right-hand side, as appropriate

Thus, $T(n) = T((n - 1)/2) + \Theta(1)$

Recursive Functions

Also, if $n = 1$, then $\mathrm{T}(1) = \Theta(1)$; thus we have to solve:

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ \mathrm{T}\left(\dfrac{n-1}{2}\right) + 1 & n > 1 \end{cases}$$

Solving this can be difficult, in general, so we will consider only special values of n

Assume $n = 2^k - 1$ where k is an integer

Then $(n - 1)/2 = (2^k - 1 - 1)/2 = 2^{k-1} - 1$

Recursive Functions

For example, searching a list of size 31 requires us to check the center

If it is not found, we must check one of the two halves, each of which is size 15

$$31 = 2^5 - 1$$

$$15 = 2^4 - 1$$

Recursive Functions

Thus, we can write

$$
\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(2^k - 1) \\
&= \mathrm{T}\left(\frac{2^k - 1 - 1}{2}\right) + 1 \\
&= \mathrm{T}(2^{k-1} - 1) + 1 \\
&= \mathrm{T}\left(\frac{2^{k-1} - 1 - 1}{2}\right) + 1 + 1 \\
&= \mathrm{T}(2^{k-2} - 1) + 2 \\
&\vdots
\end{aligned}
$$

Recursive Functions

Notice the pattern with one more step:

$$
\begin{aligned}
&= T(2^{k-1} - 1) + 1 \\
&= T\left(\frac{2^{k-1} - 1 - 1}{2}\right) + 1 + 1 \\
&= T(2^{k-2} - 1) + 2 \\
&= T(2^{k-3} - 1) + 3 \\
&\vdots
\end{aligned}
$$

Recursive Functions

Thus, in general, we may deduce that after $k - 1$ steps:

$$\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(2^k - 1) \\
&= \mathrm{T}(2^{k-(k-1)} - 1) + k - 1 \\
&= \mathrm{T}(1) + k - 1 = k
\end{aligned}$$

because $\mathrm{T}(1) = 1$

Recursive Functions

Thus, $\mathrm{T}(n) = k$, but $n = 2^k - 1$

Therefore $k = \lg(n + 1)$

Further, recall that $f(n) = \Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for $0 < c < \infty$

$$\lim_{n\to\infty} \frac{\lg(n+1)}{\ln(n)} = \lim_{n\to\infty} \frac{\frac{1}{(n+1)\ln(2)}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{(n+1)\ln(2)} = \frac{1}{\ln(2)}$$

Thus, $\mathrm{T}(n) = \Theta(\lg(n + 1)) = \Theta(\log(n))$

Exercises

- **CRLS 3.1-1: Let $f(n)$and $g(n)$be asymptotically nonnegative functions. Using the basic definition of Θ notation, prove that $\max(f(n), g(n)) = \Theta(f(n)+g(n))$**

- **CRLS 3.1-2: Show that for any real constants a and b, where $b > 0$, $(n+a)^b = \Theta(n^b)$**

- **CRLS 3.1-3: Explain why the statement, "The running time of algorithm *A* is at least $O(n^2)$," is meaningless.**

Exercises

- **CRLS 3.2-1**
- **CRLS 3.2-3**
- **CRLS 3-1**

算法分析课程组

重庆大学计算机学院

End of Section.