# Introduction to Computer Science

## Program Analysis

Ryan Stansifer

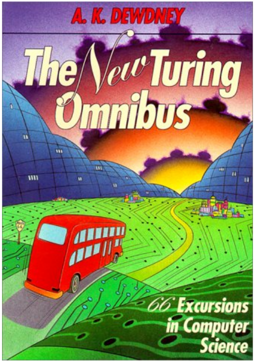
Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

`http://www.cs.fit.edu/~ryan/`

8 December 2017

# Something Related For Additional Reading

A. K. Dewdney, “Chapter 14: Time and Space Complexity,” in *The Turing Omnibus*, 1989, pages 89–95.



# Two approaches

1. Analytical. Static analysis of the program. Requires program source. (Mathematical guarantees.)
2. Empirical. Time experiments running the program with different inputs. (Scientific method.)

# Profiling

Measuring the time a program takes is difficult. Many factors influence the time: processor, OS, multitasking, input data, resolution of the clock, etc. It is difficult to predict the performance of a program in general based on timing experiments.

There is a better way using functions.

# Steps, Worse Case

The work a computer does can be measured in the number of individual instructions it executes. The work a program does can be approximated by the number of operations or steps it calls for–operations like assignment, IO, arithmetic operations and relational comparisons. The size of the steps–10 machine instructions, 100 machine instructions—does not matter in the long run.

When counting the steps of a program we always assume the worse. We assume that the program will “choose” the path that requires the most steps. This way we get an upper bound on the performance.

# Input

In common with the emperical approach, we suppose it makes sense to talk about the size of the input.

# Input

Useful programs take different steps depending on the input. So, the number of steps a program takes for some particular input does not tell us how good the program is. A bad algorithm may take few steps for some small, simple input; and a good algorithm may take many steps for some large, complex input.

# Input

Suppose we count the number of steps in terms of the *size of the input*, call it $N$. The number of steps is a function of $N$. For the program which reads $N$ numbers in order to sum them, the number of steps might be $f(N) = 2N + 1$.

What is the size of the input? Most algorithms have a parameter that affects the running time most significantly. For example, the parameter might be the size of the file to be sorted or searched, the number of characters in a string, or some other abstract measure of the size on the data set being processed.

In the long run, little differences in the number of steps do not matter, so we group functions together in larger categories to more easily see significant difference.

The number of steps a programs takes is a *function* of the size of the input.
(Different results are obtained if the size of input is measured differently. E.g., one integer unit, or $n$ bits representing an integer.)

# Asymptotic Notation

We wish to compare functions carefully by their growth. Unimportant information should be ignored, like “rounding” where

$$1,000,001 \approx 1,000,000$$

And we want the “big picture.” This means that a function $f$ may be smaller than a function $g$ for some particular values, but “in the long run” it may be larger than $g$.
Fortunately, a precise definition that captures our intuition (most of the time) is possible.

# Preliminaries

Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to non-negative integers

$$f : Z \to Z \quad \text{and} \quad g : Z \to Z$$

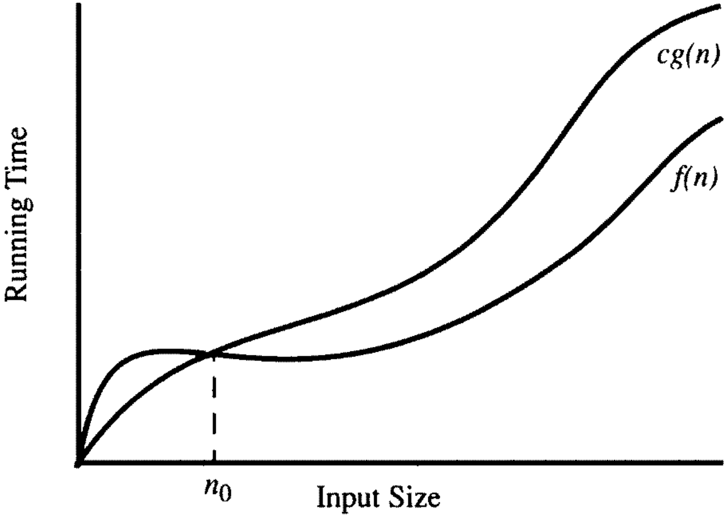
Sometimes it is more useful to generalize to functions of real value

$$f : R_{\geq 0} \to R_{\geq 0} \quad \text{and} \quad g : R_{\geq 0} \to R_{\geq 0}$$

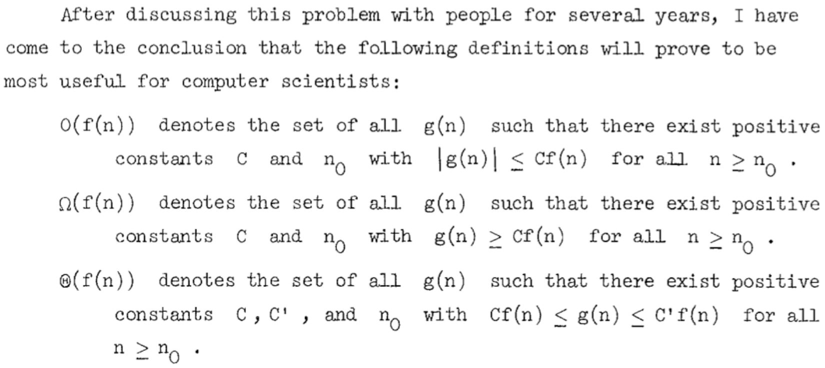
# Preliminaries

Since the input to a program cannot have negative size, and the resources consumed by a program cannot be negative, we restrict ourselves to functions whose graphs are in the first quadrant, i.e., are the range and domain are $R_{\geq 0}$.
To define the Big-Oh notation, we first give a diagram, then Knuth's original definition (in which the roles of $f$ and $g$ are swapped), and finally our definition.

# Big-Oh – $f(n)$ is $O(g(n))$



# Big-Oh



After discussing this problem with people for several years, I have come to the conclusion that the following definitions will prove to be most useful for computer scientists:

$O(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants $C$ and $n_0$ with $|g(n)| \leq Cf(n)$ for all $n \geq n_0$ .

$\Omega(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants $C$ and $n_0$ with $g(n) \geq Cf(n)$ for all $n \geq n_0$ .

$\Theta(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants $C$ , $C'$ , and $n_0$ with $Cf(n) \leq g(n) \leq C'f(n)$ for all $n \geq n_0$ .

D. Knuth, 1976.

# Categorizing functions [ f in g ]

Let $f(n)$ and $g(n)$ be functions mapping non-negative numbers to non-negative numbers.

*Big-Oh*. $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

# Categorizing functions [ f in g ]

Let $f(n)$ and $g(n)$ be functions mapping non-negative numbers to non-negative numbers.

*Big-Oh*. $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

*Big-Omega*. $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every integer $n \geq n_0$.

*Big-Theta*. $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega(f(n))$.

*Little-Oh*. $f(n)$ is $o(g(n))$ if for any $c > 0$ there is $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

*Little-Omega*. $f(n)$ is $\Omega(g(n))$ if for any $c > 0$ there is $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every number $n \geq n_0$.

# Categorizing functions

There is a family or related notions, however, $O(n)$ is the only notion required at the moment.

You are asked to commit the definition to memory now. Eventually (e.g., in Algorithms and Data Structures), you will be expected to have a deeper understanding of these notions.

$$\begin{aligned}
f(n) \text{ is } O(g(n)) &\quad\approx\quad x \leq y \\
f(n) \text{ is } \Theta(g(n)) &\quad\approx\quad x = y \\
f(n) \text{ is } \Omega(g(n)) &\quad\approx\quad x \geq y \\
f(n) \text{ is } o(g(n)) &\quad\approx\quad x < y \\
f(n) \text{ is } \omega(g(n)) &\quad\approx\quad x > y
\end{aligned}$$

The analogy is rough since some functions are not comparable, while any two real numbers are comparable.

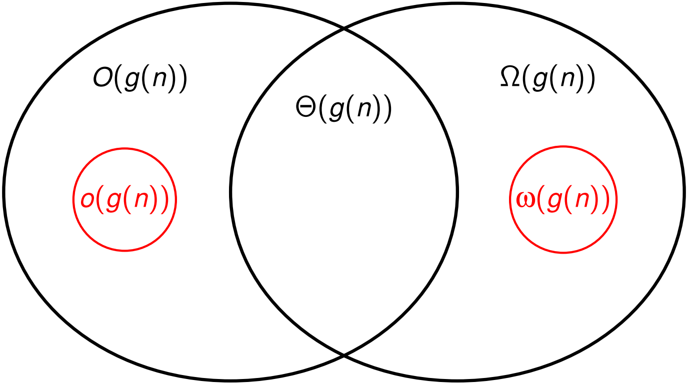
# Big-Oh and Connection to Limits [ f in g ]

Let $f(n)$ and $g(n)$ be functions mapping non-negative real numbers to non-negative real numbers.

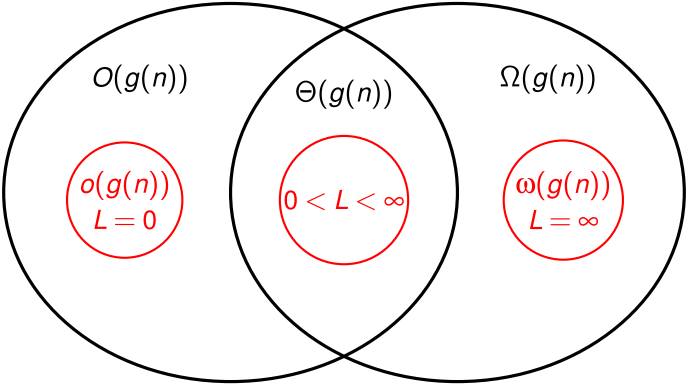
*Big-Oh*. $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

Lemma. $f(n)$ is $O(g(n))$ if (but not only if) $\lim_{n\to\infty} f(n)/g(n) = L$ where $0 < L < \infty$.
Lemma. $f(n)$ is $O(g(n))$ if, and only, if $\limsup_{n\to\infty} f(n)/g(n) = L$ where $0 < L < \infty$.

# Relationships



# Relationships



Here $L$ denotes the limit

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

.

# Example

The function $f(n) = 3 \cdot n + 17$ is $O(n)$. (Here $g(n) = n$.)
Proof. Take $c = 4$ and $n_0 = 17$. Then $f(n) = 3 \cdot n + 17 \leq c \cdot g(n)$ for every $n \geq n_0$. because $3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n$ for every $n \geq 17$.

# Example

The function $f(n) = 3 \cdot n + 17$ is $O(n)$. (Here $g(n) = n$.)
Proof. Take $c = 4$ and $n_0 = 17$. Then $f(n) = 3 \cdot n + 17 \leq c \cdot g(n)$ for every $n \geq n_0$. because $3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n$ for every $n \geq 17$.
$f(n) = 4 \cdot n + 17$ is $O(n)$?

# Example

The function $f(n) = 3 \cdot n + 17$ is $O(n)$. (Here $g(n) = n$.)
Proof. Take $c = 4$ and $n_0 = 17$. Then $f(n) = 3 \cdot n + 17 \leq c \cdot g(n)$ for every $n \geq n_0$. because $3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n$ for every $n \geq 17$.
$f(n) = 4 \cdot n + 17$ is $O(n)$?
$f(n) = 3 \cdot n + 88$ is $O(n)$?

# Using the Big-Oh Notation

The notation is bad. It is difficult to use. [The language of mathematics has (and this is quite amazing) dealt very poorly with functions. Church's lambda notation is not widely used.]
The idea is simple: a function gives rise to a collection of functions containing that function and other functions.

It is best to write

$$f(n) \textit{ is } O(g(n))$$

Some authors write $f(n) \in O(g(n))$, or even $f(n) = O(g(n))$, but I find that misleading.

# Big-Oh Math

Lemma: If $d(n)$ is $O(f(n))$, then $a \times d(n)$ is $O(f(n))$, for any constant $a > 0$. Just take $c = a \times c_1$.

Another fact: If $f(n)$ and $h(n)$ are both $O(g(n))$, then $f(n) + h(n)$ is $O(g(n))$; just take $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$.

Finally: If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $O(n^d)$.

For example, if $f(n) = an^2 + bn + c$, then it is $O(n^2)$.

# Big-Oh Math

Lemma: $n^d$ is in $O(n^{d+1})$

# Big-Oh Math

Fact: $f(n) = n$ is $O(2^n)$ because, by induction, $n < 2^n$ for all $n$.

Another fact: $2^{n+4} = 2^4 \times 2^n < (2^4 + 1) \times 2^n$, so take $c = 2^4 + 1$ and therefore, $2^{n+4}$ is $O(2^n)$.

# Important Categories of Functions

| | |
|---|---|
| $O(1)$ | constant |
| $O(\log n)$ | logarithmic |
| $O(n)$ | linear |
| $O(n \log n)$ | loglinear |
| $O(n^2)$ | quadratic |
| $O(n^3)$ | cubic |
| $O(2^n)$ | exponential |

# Intractable Problems

A problem is said to be intractable if the algorithm takes an impractical amount of time to find the solution.
Roughly speaking, we consider polynomial algorithms to be tractable and exponential algorithms to be impractical.

# Fast Growing Functions

*Observation 1: You cannot make an inefficient algorithm efficient by how you choose to implement it or what machine you choose to run it on.*

*Observation 2: It is virtually impossible to ruin the efficiency of an efficient algorithm by how you implement it or what machine you run it on.*

So, the efficiency is determined by the algorithms and data structures used in your solution. Efficiency is not significantly affected by how well or how poorly you implement the code.

# Fast Growing Functions

*The order of an algorithm is generally more important than the speed of the processor.*

Fast growing functions grow really fast. Their growth is stupefying. Don’t be misled.

Goodrich and Tamassia, Table 3.2, page 120.

# Comparing Functions

In finding a name in phone book, suppose every comparison takes one millisecond (0.001 sec).

| city | pop | linear | binary |
|---|---:|---:|---:|
| Port St. Lucie | 164,603 | 2.8 min | 0.017 sec |
| Fort Lauderdale | 165,521 | 2.8 min | 0.017 sec |
| Tallahassee | 181,376 | 3.0 min | 0.017 sec |
| Hialeah | 224,669 | 3.7 min | 0.018 sec |
| Orlando | 238,300 | 4.0 min | 0.018 sec |
| St. Petersburg | 244,769 | 4.0 min | 0.018 sec |
| Tampa | 335,709 | 5.6 min | 0.018 sec |
| Miami | 399,457 | 6.7 min | 0.019 sec |
| Jacksonville | 821,784 | 13.7 min | 0.020 sec |

# Comparing Functions

In finding a name in phone book, suppose every comparison takes one microsecond (0.001 sec).

| city | pop | linear | binary |
|---|---:|---:|---:|
| Dallas, TX | 1,299,543 | 21.7 min | 0.020 sec |
| San Diego, CA | 1,306,301 | 21.8 min | 0.020 sec |
| San Antonio, TX | 1,373,668 | 22.9 min | 0.020 sec |
| Philadelphia, PA | 1,547,297 | 25.8 min | 0.021 sec |
| Phoenix, AZ | 1,601,587 | 26.7 min | 0.021 sec |
| Houston, TX | 2,257,926 | 37.6 min | 0.021 sec |
| Chicago, IL | 2,851,268 | 47.5 min | 0.021 sec |
| Los Angeles, CA | 3,831,868 | 63.9 min | 0.022 sec |
| New York, NY | 8,391,881 | 139.9 min | 0.023 sec |

# Comparing Functions

In finding a name in phone book, suppose every comparison takes one microsecond (0.001 sec).

| city | pop | linear | binary |
|---|---:|---:|---:|
| Seoul | 10,575,447 | 2.9 hr | 0.023 sec |
| São Paulo | 11,244,369 | 3.1 hr | 0.023 sec |
| Moscow | 11,551,930 | 3.2 hr | 0.023 sec |
| Beijing | 11,716,000 | 3.3 hr | 0.023 sec |
| Mumbai | 12,478,447 | 3.5 hr | 0.024 sec |
| Delhi | 12,565,901 | 3.5 hr | 0.024 sec |
| Istanbul | 12,946,730 | 3.6 hr | 0.024 sec |
| Karachi | 12,991,000 | 3.6 hr | 0.024 sec |
| Shanghai | 17,836,133 | 5.0 hr | 0.024 sec |

# Fast Growing Functions

| $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ | |
|---:|---:|---:|---:|---:|---:|---|
| 3 | 10 | 30 | 100 | 1,000 | 1,024 | kilo |
| 4 | 20 | 80 | 400 | 8,000 | 1,048,576 | mega |
| 4 | 30 | 120 | 900 | 27,000 | 1,073,741,824 | giga |
| 5 | 40 | 200 | 1,600 | 64,000 | 1,099,511,627,776 | tera |
| 5 | 50 | 250 | 2,500 | 125,000 | 1,125,899,906,842,624 | peta |
| 6 | 60 | 300 | 3,600 | 216,000 | $1.15 \times 10^{18}$ | exa |
| 6 | 70 | 420 | 4,900 | 343,000 | $1.18 \times 10^{21}$ | zetta |
| 6 | 80 | 480 | 6,400 | 512,000 | $1.21 \times 10^{24}$ | yotta |
| 6 | 90 | 540 | 8,100 | 729,000 | $1.24 \times 10^{27}$ | |
| 6 | 100 | 600 | 10,000 | 1,000,000 | $1.27 \times 10^{30}$ | |

# Algorithms Have CHanged the World

- FFT
- Barnes-Hut

# Categorizing Programs

Compute $\sum_{i=1}^{n} i$

Algorithm 1 – $O(n)$

```
final int n = Integer.parseInt (args[0]);
int sum = 0;
for (int count=1; count<=n; i++) {
   sum += count;
}
```

Algorithm 2 – $O(1)$

```
final int n = Integer.parseInt (args[0]);
int sum = (n*(n+1))/2;
```

Give a Big-Oh analysis in terms of *N* of the running time for each of the following program fragments:

```
for (int i=1; i<N; i++) {
    sum++;
}
```

Give a Big-Oh analysis in terms of $N$ of the running time for each of the following program fragments:

```
for (int i=1; i<N; i++) {
    sum++;
}
```

$O(N)$

```
for (int i=1; i<N; i+=2) {
    sum++;
}
```

Give a Big-Oh analysis in terms of $N$ of the running time for each of the following program fragments:

```
for (int i=1; i<N; i++) {
    sum++;
}
```

$O(N)$

```
for (int i=1; i<N; i+=2) {
    sum++;
}
```

$O(N/2) = O(N)$

```
for (int i=1; i<N; i++) {
    for (int j=1; j<N; j++) {
        sum++;
    }
}
```

```
for (int i=1; i<N; i++) {
    for (int j=1; j<N; j++) {
        sum++;
    }
}
```

$O(N^2)$

```
for (int i=1; i<10; i++) {
    O(N) steps in loop
}
```

```
for (int i=1; i<10; i++) {
    $O(N)$ steps in loop
}
```

$$O(10N) = O(N)$$

```
for (int i=1; i<N; i++) {
    sum++;
}
for (int j=1; j<N; j++) {
    sum++;
}
```

```
for (int i=1; i<N; i++) {
    sum++;
}
for (int j=1; j<N; j++) {
    sum++;
}
```

$2O(N) = O(N)$

```
for (int i=1; i<N; i++) {
    sum++;
}
for (int j=1; j<N; j++) {
    sum++;
}
```

$2O(N) = O(N)$

```
for (int i=1; i<=N; i++) {
    for (int j=1; j<=N*N; j++) {
        for (int k=1; k<=j; k++) {
            sum++;
        }
    }
}
```

```
for (int i=1; i<N; i++) {
    sum++;
}
for (int j=1; j<N; j++) {
    sum++;
}
```

$2O(N) = O(N)$

```
for (int i=1; i<=N; i++) {
    for (int j=1; j<=N*N; j++) {
        for (int k=1; k<=j; k++) {
            sum++;
        }
    }
}
```

$N \times \sum_{k=1}^{N^2} k = O(N \times (N^2 \times (N^2 - 1)/2)) = O(N^5)$

```
for (int i=1; i<N; i*=2) {
   sum++;
}

while (N>1) {
   N = N/2;
   /* O(1) */
}
```

```
for (int i=1; i<N; i*=2) {
    sum++;
}

while (N>1) {
    N = N/2;
    /* O(1) */
}
```

$O(\log N)$

# Categorizing Programs

Compute $\lceil \log n \rceil$

Algorithm 1 – $O(\log n)$

```
for (lgN=0; Math.pow(2,lgN)<n; lgN++);
```

Algorithm 2 – $O(\log n)$

```
for (lgN=0; n>0; lgN++, n/=2);
```

Algorithm 3 – $O(\log n)$

```
for (lgN=0,t=1; t<n; lgN++, t += t);
```

# Some Recursive Patterns

```
public static void g (int N) {
  if (N==0) return;
  g (N/2); // half the amount work
}
```

# Some Recursive Patterns

```
public static void g (int N) {
  if (N==0) return;
  g (N/2); // half the amount work
}
```

$O(\log N)$ as in binary search

`basic_algorithms/RecursiveBinary.java`

`basic_algorithms/Binary.java`

`basic_algorithms/GenericBinary.java`

Towers of Hanoi??

```
public static void g (int N) {
  if (N==0) return;
  g (N/2);   // half the amount work
  g (N/2);   // not the same work
  /* O(N) */
}
```

```
public static void g (int N) {
  if (N==0) return;
  g (N/2);    // half the amount work
  g (N/2);    // not the same work
  /* O(N) */
}
```

$O(N \log N)$ as in merge sort. This pattern is asociated with the divide-and-conquer strategy for problem solving. `sort/Merge.java`

# Quick Sort

Looks like the same pattern as merge sort, but it different. This is subtle and important.

`sort/Quick.java`

```
public static void f (int N) {
  if (N==0) return;
  f (N-1);
  f (N-1);
  /* O(1) */
}
```

```
public static void f (int N) {
  if (N==0) return;
  f (N-1);
  f (N-1);
  /* O(1) */
}
```

$O(2^N)$

`basic_algorithms/TowersOfHanoi.java` Towers of Hanoi at Wiki

# Problems

There is likely more than one algorithm to solve a problem.

**Minimum Element in an Array.** Given an array of $N$ items, find the smallest item.
**Closest Points in the Plane.** Given $N$ points in a plane, find the pair of points that are closest together.
**Co-linear Points in the Plane.** Given $N$ points in a plane, determine if any three form a straight line.

# Prefix Averages

Two algorithms to solve a simple problems.

`oh/PrefixAverages.java` Java program

# Maximum Contiguous Subsequence Sum

MAXIMUM CONTIGUOUS SUBSEQUENCE SUM PROBLEM. Given (possibly negative) integers $a_1$, $a_2$, …, $a_n$, find (and identify the sequence corresponding to) the maximum value of $\sum_{k=i}^{j} a_k$. The maximum contiguous subsequence sum is zero if all the integers are negative.

For example, if the input is $\{-2, \mathbf{11}, \mathbf{-4}, \mathbf{13}, -5, 2\}$, then the answer is 20 which corresponds to the contiguous subsequence encompassing elements 2 through 4.

Weiss, Section 5.3, page 153.

# Maximum Contiguous Subsequence Sum

The obvious $O(n^3)$ algorithm: for every potential staring element of the subsequence, and for every potential ending element of the subsequence, find the one with the maximum sum.

# Maximum Contiguous Subsequence Sum

Since $\sum_{k=i}^{j+1} a_k = (\sum_{k=i}^{j} a_k) + a_{j+1}$, the sum of the subsequence $a_i$, $a_{i+1}$, …, $a_{j+1}$ can be computed easily (without a loop) from the sum of $a_i$, $a_{i+1}$, …, $a_j$.

# Maximum Contiguous Subsequence Sum

Theorem. Let $a_k$ for $i \leq k \leq j$ be any subsequence with $\sum_{k=i}^{j} a_k < 0$. If $q > j$, then $a_k$ for $i \leq k \leq q$ is *not* a maximum contiguous subsequence.

Proof. The sum of the subsequence $a_k$ for $j+1 \leq k \leq q$ is larger.

# Maximum Contiguous Subsequence Sum

`oh/MaxSubsequenceSum.java` Java program

# Dynamic Programming

See Sedgwick and Wayne.

# Math Review

$$\log_b a = c \quad \text{if} \quad a = b^c$$

Nearly always we want the base to be 2.

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

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Lots of discrete steps

- $\lceil x \rceil$ the largest integer less than or equal to $x$.
- $\lfloor x \rfloor$ the smallest integer less than or equal to $x$.