Topic 2:

Algorithm Analysis

What are Some Algorithms You've Learned?

Desirable Algorithm Characteristics

A good algorithm . . .

How Can We Measure Problem Size?

Definition: Instance Characteristic

. . . .

. . . .

Example(s):

Measuring the Speed of an Algorithm

Why? To compare its speed to that of other algorithms.

Two approaches:

Step–Counting (a.k.a. Operation Counting)

A simple, imprecise (but illuminating) technique. The approach:

Example: The mean of an array's values (1 / 4)

Here's the first algorithm we will be step–counting:

```
double sum = 0;
for (int i=0; i<n; i++) {
    sum = sum + list[i];
}
mean = sum / n;
```

First question: What is/are the instance characteristic(s)?

Detour: How to Step–Count a For Loop (1 / 2)

First, imagine we initialize an operation counter (`o = 0;`).

Second, imagine we augment the code with `o++;`

statements to count the loop's operations:

for (**initialization** ; **condition** ; **increment**) {

loop body

}

Detour: How to Step–Count a For Loop (2 / 2)

Summary: To step–count a `for` loop, count:

- The initialization expression before the loop
- True evaluations of the loop condition within the loop body
- False evaluation of the loop condition after the loop
- The increment expression within the loop body

Example: The mean of an array's values (2 / 4)

(1) Augment the algorithm with operation counts, and

(2) Estimate executions of selections and iterations

```
double sum = 0;
for (int i=0; i<n; i++) {
    sum = sum + list[i];
}
mean = sum / n;
```

Example: The mean of an array’s values (3 / 4)

(3) Remove/Ignore the algorithm’s statements

Example: The mean of an array’s values (4 / 4)

(4) Produce a step–count expression in terms of the algorithm’s instance characteristics(s)

```
o++;
o++;
iterate n times:
    o++;
    o++; o++; o++; o++;
    o++;
o++;
o++; o++;
```

How to Step–Count an If Statement

Three possible approaches:

```
if ( condition ) {
    body
}
```


How to Step–Count an If–Else Statement

Per execution, we do either the ‘then’ part or the ‘else’ part, but not both.

Being pessimistic, we . . .

```
if ( condition ) {
    ‘then’ part
} else {
    ‘else’ part
}
```

Example: Min & Max of an array's values (1 / n)

. . . pessimistically!

Here's Version 1:

```
double min, max;
min = max = list[0];

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


Example: Min & Max of an array's values (2 / n)

Here's one possible pessimistic step–counting result:

```
o++; o++; o++; o++;
o++;
iterate n-1 times:
     o++;
     o++; o++; o++; o++;
     o++; o++;
     o++;
o++;
```

Example: Min & Max of an array’s values (3 / n)

Version 2: What if we partition pairs of values?

2	5	9	6	···	

Example: Min & Max of an array’s values (4 / n)

Code for Version 2:

```
int low = 0, high = n;
int i;

for (i=0; i<n-1; i+=2) {
  if (list[i] < list[i+1]) {
    candidates[low++] = list[i];
    candidates[high--] = list[i+1];
  } else {
    candidates[low++] = list[i+1];
    candidates[high--] = list[i];
  }
}

if (i == n-1) {
  candidates[low++] = list[i];
  candidates[high--] = list[i];
}
min = candidates[0];
for (int j=1; j<low; j++) {
    if (candidates[j] < min)
        min = candidates[j];
}

max = candidates[high+1];
for (int k=high+2; k<n+1; k++) {
    if (candidates[k] > max)
        max = candidates[k];
}
```

Example: Min & Max of an array's values (5 / n)

Saving time, here's a summary of Version 2's step–counting:

			3	Lines 1–4 (before 1st loop)
+	18(n/2)			Lines 4–12 (partition loop)
+			18	Lines 13–20 (between loops)
+	6(n+1)/2			Lines 20–23 (min loop)
+			7	Lines 24–26 (between loops)
+	7(n+1)/2			Lines 26–29 (max loop)
+			2	Line 30 (after max loop)
≈	15.5n	+	36.5	# of steps in terms of `n`

Example: Min & Max of an array's values (6 / n)

Version 3: Combine partitioning and min–max finding.

```
min = max = list[0];

for (int i=1; i<n-1; i+=2) {
    if (list[i] < list[i+1]) {
        if (list[i] < min)
            min = list[i];
        if (list[i+1] > max)
            max = list[i+1];
    } else {
        if (list[i+1] < min)
            min = list[i+1];
        if (list[i] > max)
            max = list[i];
    }
}

if (i == n-1) {          // handle extra single value
    if (list[i] < min)
        min = list[i];
    if (list[i] > max)
        max = list[i];
}
```

Example: Min & Max of an array's values (7 / n)

Summary of Version 3's pessimistic step–counting:

	5	Lines 1–3 (before the loop)
+	14(n-1)/2	Lines 3–15 (the loop)
+	10	Lines 16–22 (after the loop)
$\approx$	7n + 8	# of steps in terms of n

Key Comparisons: Focused Step–Counting

Step–counting can be *slightly* tedious.

Instead, we can count only operations of special interest.

Definition: Key Comparison

. . . .

. . . .

Example(s):

Key Comparisons in the Min/Max Algorithms

Adding approximate key comparisons to our results:

Version	Operations	Key Comparisons
1	8n - 2	$\approx$ 2n
2	15.5n + 36.5	$\approx \frac{3}{2}$ n
3	7n + 8	$\approx \frac{3}{2}$ n

Key comparisons can differ from overall step–counts.

Code Profiling (1 / 4)

Definition: Code Profiling

. . . .

. . . .

Code Profiling (2 / 4)

To generate `jfr` data from a program, execute your Java program with these flags:

- `-XX:+UnlockDiagnosticVMOptions`
- `-XX:+DebugNonSafepoints`
- `-XX:StartFlightRecording=duration=[S]s,filename=[N].jfr`

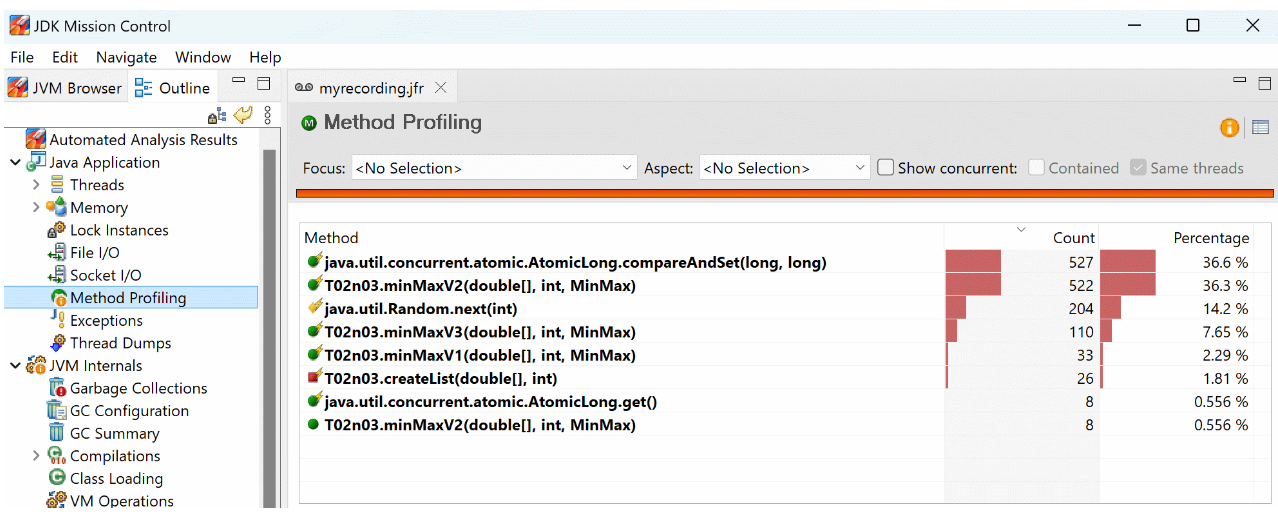
where `[S]` is the measurement duration in seconds, and `[N]` is the file name of the `jfr` recording output file.

Example:

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints
-XX:StartFlightRecording=duration=60s,filename=fr.jfr T02n03
1000000
```


Code Profiling (3 / 4)

Here's some `jmc` profiling output from `T02n03.java`:



Code Profiling (4 / 4)

Notes:

- `jfr`'s sampling frequency isn't high; the results are coarse
- Commercial profilers are more sophisticated
  - E.g., can do statement–level profiling
- Instrumenting code does slow it down; amount varies
- Most popular languages have available profilers
  - You may learn about tools such as `gprof`, `valgrind`, and `gcov` in CSc 352
- All of these require writing code to be analyzed!

Another Option: Execution Timing

We can 'click a stopwatch' before and after code executes:

```
start = System.nanoTime();
// call or place your code here
seconds = (System.nanoTime() - start) / 1_000_000_000;
```

So why not just do this? Some challenges:

- Wall–clock time vs. actual CPU time
- nanosecond precision, maybe not nanosecond resolution
- Many programs are are very long–running
- Code profilers do more than executing profiling
- Still have to write the code!

Questions about an Algorithm (1 / 2)

What do we want to know about an algorithm's efficiency before we adopt it?

- Will it find correct answers in a reasonable amount of time?
- Is it better than this other algorithm we're considering? (If so, under which circumstances?)
- How much RAM does this algorithm need?
- How much slower will it be if the problem size doubles?

Questions about an Algorithm (2 / 2)

Now that we know what we want to know . . .

- Can we answer those questions without having to code the algorithm and test it on sample data?
- How can I communicate those answers to other people?
- Can I do that communication clearly with math?

Asymptotic Analysis

Remember our step–counting results? Functions of n!

Definition: Asymptotic Analysis

. . . .

. . . .

"Big–O" Notation (1 / 3)

from P. Bachmann's "Analytische Zahlentheorie," 1892.

"Ordnung" is German for "order."

The idea: Have an approximate function growth rate notation.

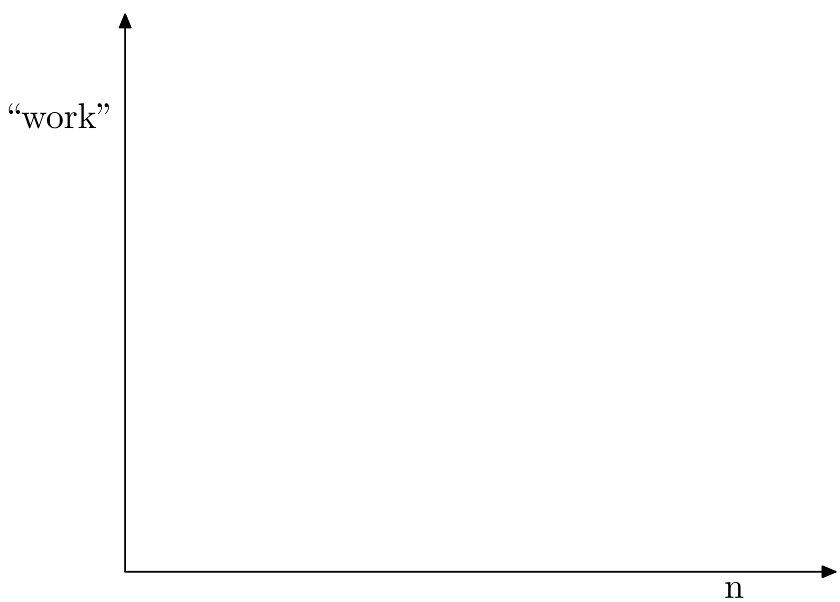
Example(s):

“Big–O” Notation (1 / 4)

Definition: “Big–O” Notation

“Big–O” Notation (2 / 4)

Looking at a plot of the functions really helps!



“Big–O” Notation (3 / 4)

Example(s): Show that $7n + 8$ is $O(n)$.

We need constants $c > 0$ and $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every integer $n \geq n_0$.

“Big–O” Notation (4 / 4)

Conjecture: $7n + 8 \leq 8n, \forall n \geq 8$

Worried that n isn't an upper–bound to $7n + 8$?

Don't be! Here's why:

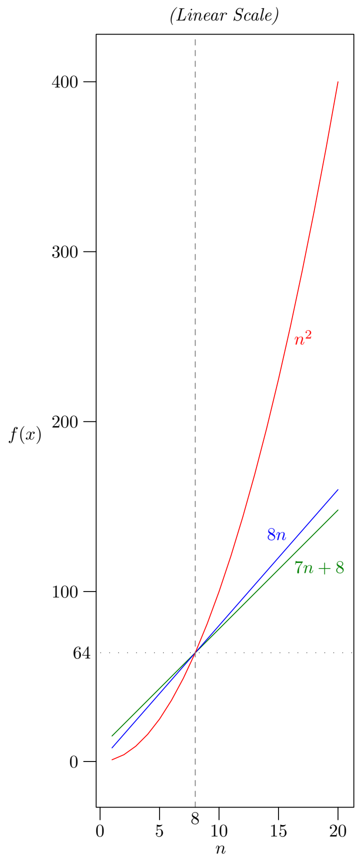
- The definition says that $8n$ is the upper–bound, not n.
- And then, only when $n \leq 8$ (that is, when n is large).

Worried that $7n + 8$ is $O(n^2)$, too?

You should be . . . and you shouldn't be!

You should be concerned, because:

You shouldn't be concerned, because:



Common Algorithm Function Classes (1 / 2)

Function	Common Name	Example Algorithm

Common Algorithm Function Classes (2 / 2)

Notes on the table:

A Helpful “Big–O” Theorem (1 / 3)

Conjecture: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$, then $f(n)$ is $O(n^m)$.

A Helpful “Big–O” Theorem (2 / 3)

Conjecture: (proof continues!)

A Helpful "Big–O" Theorem (3 / 3)

A concrete example will help that theorem make sense.

Example(s):

Flashback to the Min/Max Algorithms

Recall their step–counting functions:

Version 1: $f(n) = 8n - 2$

Version 2: $f(n) = 15.5n - 36.5$

Version 3: $f(n) = 7n + 8$

What can we say about them in terms of Big–O?

The point:

Beyond Big–O

Some issues with Big–O:

$O()$, $o()$; what begins with 'O'? (1 / 3)

Apologies to Theodor S. Geisel

Starting Idea: If we create an upper–bound that *must* be loose, Big–O is free to be used as a tight upper–bound.

Definition: Little–o ($o()$)

Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $g : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$.

$O()$, $o()$; what begins with 'O'? (2 / 3)

Example(s):

Is $7n + 8 \in o(n)$?

$O()$, $o()$; what begins with 'O'? (3 / 3)

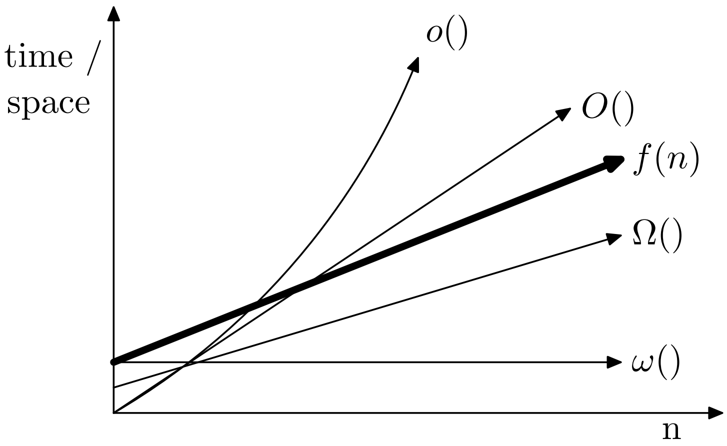
Let's try an asymptotically greater upper–bound.

Example(s):

Is $7n + 8 \in o(n^2)$?

What about Lower Bounds?

Easy! We can create mirror–images (sort of) of $O()$ and $o()$.



Big–Omega ($\Omega()$)

Big–Omega is the 'mirror–image' of Big–O.

Definition: Big–Omega ($\Omega()$)

> Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $g : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$. We say that $f(n)$ is $\Omega(g(n))$ if there exists a real constant $c > 0$ and there exists an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every integer $n \geq n_0$.

Little–Omega ($\omega()$)

Little–Omega is the 'mirror–image' of Little–O.

Definition: Little–omega ($\omega()$)

Let $f : \mathbb{Z}^+ \to \mathbb{R}^+$ and $g : \mathbb{Z}^+ \to \mathbb{R}^+$. We say that $f(n)$ is $\omega(g(n))$ if **for any** real constant $c > 0$, **there exists** an integer constant $n_0 \geq 1$ such that $f(n)$ $\boxed{>}$ $c \cdot g(n)$ for every integer $n \geq n_0$.

Big–Theta (Θ): The Big Squeeze

To ensure that we know how $f(n)$ behaves, we need to guarantee that our upper– and lower–bound are both tight.

Definition: Big–Theta (Θ)

“But why do people still use Big–O?”

Two reasons:

- We use Big–O as a tight upper–bound, and so its $g()$ is the same as Big–Theta’s $g()$
- Big–O is challenging enough to explain; the concept of Big–Omega is also needed to define Big–Theta.

Big–O and Friends: Comparison / Summary

Definition	? $c > 0$	? $n_0 \geq 1$	$f(n)$? $c \cdot g(n)$
$o()$	$\forall$	$\exists$	$<$
$O()$	$\exists$	$\exists$	$\leq$
$\Theta()$	$\ldots$	$\ldots$	$\ldots$
$\Omega()$	$\exists$	$\exists$	$\geq$
$\omega()$	$\forall$	$\exists$	$>$

Some Properties of Asymptoticity (1 / 2)

First two (well, really three):

Symmetry of Θ:

$f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$

Some Properties of Asymptoticity (2 / 2)

Last two:

Example(s):

Analyzing Subdivided Algorithms

Algorithms are often multi–part. We can analyze the parts separately, but how do we combine those results?

Explaining Big–O et al. Using Limits (1 / 3)

A wee bit of calculus background:

(1) Derivatives of Polynomials

If $f(n) = c \cdot n^r$, then $f'(n) = c \cdot r \cdot n^{r-1}$

Example(s):

(2) L'Hôpital's Rule

If the limits of $f(n)$ and $g(n)$ are $\in \{-\infty, 0, \infty\}$, and $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$.

Explaining Big–O et al. Using Limits (2 / 3)

Defining Big–O with limits:

If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, where $0 \le c < \infty$, then $f(n) \in O(g(n))$.

Example(s):

Explaining Big–O et al. Using Limits (3 / 3)

There are similar definitions for the rest:

$O()$: If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, where $0 \le c < \infty$, then $f(n) \in O(g(n))$.

$o()$: If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then $f(n) \in o(g(n))$.

$\Omega()$: If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, where $0 < c \le \infty$, then $f(n) \in \Omega(g(n))$.

$\omega()$: If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$, then $f(n) \in \omega(g(n))$.

$\Theta()$: If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, where $0 < c < \infty$, then $f(n) \in \Theta(g(n))$.

Analysis of Recursive Algorithms

Step–Counting doesn't work well with recursion . . . but recurrence relations do!

Example(s):

Recurrence Relations in Algorithm Analysis

We have seen that polynomials describe the work performed by iterative algorithms.

Solving RRs Using “Find the Pattern” (1 / 4)

Consider the classic: Computing factorials recursively.

```
public long factorial (long n)
{
    if (n == 0) return 1;
    else        return (n * factorial(n - 1));
}
```


Solving RRs Using “Find the Pattern” (2 / 4)

Time to find the pattern!

$$F(0) = c$$

$$F(n) = F(n-1) + k$$

Solving RRs Using “Find the Pattern” (3 / 4)

Next step: Generalize the expression sequence (find the pattern!).

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

Solving RRs Using “Find the Pattern” (4 / 4)

Recall: $F(0) = c$

Question: In $F(n) = F(n-a) + ak$, how large can a become?

But . . . Were Our Assumptions Correct?

Conjecture: The solution to the recurrence $F(n) = F(n-1)+k$ (with initial condition $F(0) = c$) is $F(n) = c + kn$.

Summary of the “Find the Pattern” Process

1. Determine the work required for the base and general cases of the given recursive algorithm.
2. Generate a few more equivalent recurrences for the work required in the general case.
3. Find the pattern within those expressions.
4. Create an equivalent closed–form (non–recurrence) expression in terms of the instance characteristic(s).
5. Prove that your closed–form expression is correct.
6. Determine the order of the algorithm.

Recursively Find the Max Value in a List (1 / 7)

Given an array or list of values, what is the largest value?

14	26	53	23	12	36	41	17	10	42	19	39

Instead of linear search, let's use this algorithm:

Recursively Find the Max Value in a List (2 / 7)

That algorithm works . . . but how much work does it perform?

Step 1 Determine the base and general cases.

The smallest useful list contains one value. Work required:

For a list of n items, the work required is:

Aside: “Yeah, about that $n/2$ assumption . . . ”

If the list size is odd, $\frac{n}{2}$ isn’t an integer!

The true recurrence relation for this algorithm is:

Recursively Find the Max Value in a List (3 / 7)

Step 2 Generate more equivalent recurrences.

$$\begin{aligned} T(1) &= c \\ T(n) &= 2T(n/2) + k \end{aligned}$$

Recursively Find the Max Value in a List (4 / 7)

Step 3 Find the pattern! Our recurrences are:

$$\begin{aligned} T(n) &= 2T(n/2) + k \\ T(n) &= 4T(n/4) + 3k \\ T(n) &= 8T(n/8) + 7k \end{aligned}$$

Recursively Find the Max Value in a List (5 / 7)

Step 4 Create a equivalent closed–form expression.

$$T(n) = 2^i T(n/2^i) + (2^i - 1)k, \text{ where } i \in \mathbb{Z}^+.$$

What must the relationship be between n and i to reach $T(1)$?

Recursively Find the Max Value in a List (6 / 7)

Step 5 Prove that the closed–form expression is correct.

Recursively Find the Max Value in a List (7 / 7)

Step 6 Determine the order of the algorithm.

$$T(n) = (c + k)n - k$$

Some Great Theorem Names

- The Fundamental Theorem of Arithmetic
- Fermat's Last Theorem
- Lickorish twist theorem
- The Squeeze Theorem (a.k.a. The 'Two Cops and a Drunk' Theorem)
- The Ham Sandwich Theorem
- The Hairy Ball Theorem

But next for us, it's . . .

The Master Theorem (1 / 3)

(a.k.a. The Master Method)

Given a recurrence of the form $T(n) = a \cdot T(n/b) + c \cdot n^d$, where

- $T(n)$ is an increasing function,
- $n = b^k$, where $k \in \mathbb{Z}$ and $k > 0$,
- a is a real and ≥ 1,
- b is an integer and > 1,
- c is a real and > 0, and
- d is a real and ≥ 0, then

The Master Theorem (2 / 3)

Example(s):

Consider the recurrence $T(n) = 2T(n/2) + n$.

The Master Theorem (3 / 3)

Remember our 'max value' recurrence? $T(n) = 2T(n/2) + k$

Does it fit the form of the Master Theorem?

The Maximum Contiguous Subsequence Sum Problem (1 / 9)

(see Jon Bentley's "Programming Pearls")

Given integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$, where $0 \leq i, j \leq n$. If all values in the sequence are negative, the sum is 0 (subsequences may be empty).

Example(s):

What is the MCSS of the list $[-6, 4, 2, -3, 4, -4, 5, -3, 2]$?

The Maximum Contiguous Subsequence Sum Problem (2 / 9)

Algorithm #1: Exhaustive Search (credit: Ulf Grenander)

Idea: Compute sums of all possible subranges $i..j$.

```
public static int maxSubsequenceSumV1 (int[] list, int n)
{
    int thisSum, maxSum = 0;

    for (int i=0; i<n; i++) {
        for (int j=i; j<n; j++) {
            thisSum = 0;
            for (int k=i; k<=j; k++) {
                thisSum += list[k];
            }
            if (thisSum > maxSum) maxSum = thisSum;
        }
    }
    return maxSum;
}
```

The Maximum Contiguous Subsequence Sum Problem (3 / 9)

Analysis of Algorithm #1: Abbreviated step–counting!

Each loop can execute a maximum of n times. More precisely:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1 = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)$$

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

$$= \frac{n^3+3n^2+2n}{6}$$

$$\in O(n^3)$$

The Maximum Contiguous Subsequence Sum Problem (4 / 9)

Algorithm #2: Smarter Exhaustive Search (credit: Ulf Grenander)

Idea: Consider each lower endpoint (i) just once.

```
public static int maxSubsequenceSumV2 (int[] list, int n)
{
    int thisSum, maxSum = 0;

    for (int i=0; i<n; i++) {
        thisSum = 0;
        for (int j=i; j<n; j++) {
            thisSum += list[j];
            if (thisSum > maxSum) maxSum = thisSum;
        }
    }
    return maxSum;
}
```

The Maximum Contiguous Subsequence Sum Problem (5 / 9)

Analysis of Algorithm #2: Let's show the details this time.

Our nested–sum step–counting expression is: $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1$

$$\begin{aligned}
\sum_{j=i}^{n-1} 1 &= [(n-1)-i]+1 = n-i \\
\sum_{i=0}^{n-1} (n-i) &= n+(n-1)+\cdots+(n-(n-1)) \\
&= \sum_{k=1}^{n} k \\
&= \frac{n(n+1)}{2} \\
&\in O(n^2)
\end{aligned}$$

The Maximum Contiguous Subsequence Sum Problem (6 / 9)

Algorithm #3: Divide and Conquer (credit: Michael Shamos)

Example(s):

Consider splitting the list in half:

$[-6, 4, 2, -3, 4]$ and $[-4, 5, -3, 2]$.

The Maximum Contiguous Subsequence Sum Problem (7 / 9)

Analysis of Algorithm #3: Do we need the code? Nope!

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2T(n/2) + n && \text{the straddle case is linear} \\
&= 2^k T(n/2^k) + kn && \text{where } k = \log_2 n \\
&= nT(1) + n \log_2 n \\
&\in O(n \log_2 n)
\end{aligned}
$$

The Maximum Contiguous Subsequence Sum Problem (8 / 9)

Algorithm #4: Work Smarter, Not Harder! (credit: Jay Kadane)

Idea: Extend the fixed–endpoint straddling idea to discard subsequences that have a non–positive sum.

Example(s):

Consider our list again: $[-6, 4, 2, -3, 4, -4, 5, -3, 2]$.

The Maximum Contiguous Subsequence Sum Problem (9 / 9)

Analysis of Algorithm #4:

This is very obviously linear, but let's look at the sum anyway:

$$\sum_{j=0}^{n-1} 1 = [(n-1)-0]+1 = n \in O(n)$$