Summations and Recurrences

Benjamin Cosman, Patrick Lin and Mahesh Viswanathan

Fall 2020

TAKE-AWAYS

- Closed forms for basic summations:
  - For an arithmetic sequence $a_1, \ldots, a_n$ with common difference d, the corresponding arithmetic series is $\sum_{i=1}^{n} a_i = \frac{n(a_1+a_n)}{2} = \frac{n(2a_i+(n-1)d)}{2}$.
    As a special case, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
  - For a geometric sequence $a_1, \ldots, a_n$ with common ratio r, the corresonding geometric series is $\sum_{i=1}^{n} a_i = \frac{a_1(1-r^n)}{1-r}$.
    Equivalently, $\sum_{i=0}^{n-1} ar^i = \frac{a(1-r^n)}{1-r}$.
- Recurrences
  - A recurrence is a function defined in terms of itself.
  - Finding closed forms for recurrences can be done by unrolling:
    a) "Unroll" three or four times
    b) **Predict** the form obtained after unrolling k times
    c) Find the value of k that matches the base case(s)
    d) Substitute the base case(s) and simplify
    e) Sanity check on a few values, or, ideally, prove correctness by induction
  - For special recurrences of the form $T(n) = aT(\frac{n}{b}) + f(n)$ (a recursive calls and $f(n)$ extra work), we can use the recursion tree method. The root of the tree for $T(n)$ has value $f(n)$ and a children, each of which is the root of a (recursively constructed) recursion tree for $T(\frac{n}{b})$.
    a) Draw the first few layers of the recursion tree
    b) **Predict** the total extra work at each (internal) level
    c) Compute the number of internal levels as well as the number of leaves
    d) Add up the total work at each internal level, plus the work at the leaves
    e) Simplify sums and logarithms to obtain a closed form
    f) Sanity check on a few values, or, ideally, prove correctness by induction

Summations

Consider the summation $\sum_{i=1}^{n} i^2$. We will consider this summation in two ways, both related to Algorithms Analysis that we will consider in more generality next week.

First, consider how we might implement this summation in code. The most simple way is via a for-loop, shown in the following piece of pseudocode:

```
sum-of-squares(n):

s = 0
for i from 1 to n:
  s = s + (i * i)
return s
```

For now, let us just count the number of explicit arithmetic operations, i.e., the number of times an instance of `+`, `-`, `*`, or `/` in the block of pseudocode gets executed. In each iteration of the for-loop, we see one one addition and one multiplication, for a total of $2n$ explicit arithmetic operations.

Question 1. Can we do better? That is, can we compute this sum with significantly fewer explicit arithmetic operations?

While we ponder that question, let us consider the following:

```
unknown-function(n):

1.  s = 0
2.  for i from 1 to n:
3.    for j from 1 to i:
4.      for k from 1 to i:
5.        s = s + k
6.  return s
```

Again, let us count the number of explicit arithmetic operations. In each iteration of the for-loop at line 4, there is one addition, so in each iteration of the for-loop at line 3, there are i additions. As a result, in each iteration of the for-loop at line 2, there are i^2 additions. Summing over the iterations of the for-loop in line 2, we find that the number of additions is $\sum_{i=1}^{n} i^2$.

Question 2. Besides being somewhat painful to compute by hand, formulas involving summations tend to not lend themselves easily to intuition about their values. So given a value of n, is there a simple way to count the number of explicit arithmetic operations in `unknown-function` without needing to execute `sum-of-squares`?

The answer to both questions comes in the form of the identity $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, which can be verified via induction.

This identity allows us to replace `sum-of-squares` with the following,[1] which consists of two additions, three multiplications, and one division.

[1] Some modern compilers actually do make these optimizations when they are able to. For example, search for loop or summation optimizations in LLVM.

```
sum-of-squares-v2(n):

1.  return n * (n+1) * (2*n+1) / 6
```

In this section, we will see how to convert some standard summations into a ***closed form***, which is an equivalent expression that does not involve any summations.

Example 1. Our first example will be that of an ***arithmetic series***. An *arithmetic progression* is a sequence $a_1, a_2, \ldots, a_n$ of the form $a_{i+1} = a_i + d$ for $1 \le i < n$, where d is some fixed real number. An *arithmetic series* is a sum of the form $\sum_{i=1}^{n} a_i$ where $a_1, \ldots, a_n$ is an arithmetic progression.

We will find a closed form for an arithmetic series in terms of the *length* n, the *initial term* a_1 and the *common difference* d.

First, observe that $a_1 = a_1 + (1-1)d$, and if $a_i = a_1 + (i-1)d$, then $a_{i+1} = a_i + d = (a_1 + (i-1)d) + d = a_1 + id$. So by induction, $a_i = a_1 + (i-1)d$ for $1 \le i \le n$.

Similarly, we can write $a_n = a_n - 0d$, and if $a_{n-i} = a_n - id$, then $a_{n-(i+1)} = a_{n-i} - d = (a_n - id) - d = a_n - (i+1)d$. So by induction, $a_{n-i} = a_n - id$ for $0 \le i < n$.

We will now put these two observations together using a trick commonly attributed to Gauss. Let $S = \sum_{i=1}^{n} a_i$. The two observations above allow us to rewrite S as

$$S = a_1 + (a_1 + d) + (a_1 + 2d) + \cdots + (a_n - 2d) + (a_n - d) + a_n.$$

We will now add S to itself, but written "backwards":

$$\begin{array}{rcccccccccccc} S = & a_1 & + & (a_1+d) & + & (a_1+2d) & + \cdots + & (a_n-2d) & + & (a_n-d) & + & a_n \\ +\,S = & a_n & + & (a_n-d) & + & (a_n-2d) & + \cdots + & (a_1+2d) & + & (a_1+d) & + & a_1 \\ \hline 2S = & (a_1+a_n) & + & (a_1+a_n) & + & (a_1+a_n) & + \cdots + & (a_1+a_n) & + & (a_1+a_n) & + & (a_1+a_n) \end{array}$$

so $S = \frac{n(a_1+a_n)}{2} = \boldsymbol{\frac{n(2a_1+(n-1)d)}{2}}$.

For the special case $a_1 = d = 1$, $\sum_{i=1}^{n} i = \frac{n(2+(n-1))}{2} = \boldsymbol{\frac{n(n+1)}{2}}$.

Example 2. Our next example will be that of a ***geometric series***. A *geometric progression* is a sequence $a_1, \ldots, a_n$ of the form $a_{i+1} = a_i r$ for $1 \le i < n$, where r is some fixed real number. A *geometric series* is a sum of the form $\sum_{i=1}^{n} a_i$ where $a_1, \ldots, a_n$ is a geometric progression.

We will find a closed form for a geometric series in terms of the *length* n, the *initial term* a_1, and the *common ratio* r.

Observe that $a_1 = a_1 \cdot r^0$, and if $a_i = a_1 r^{i-1}$, then $a_{i+1} = a_i r = (a_1 r^{i-1})r = a_1 r^i$. So by induction, $a_i = a_1 r^{i-1}$ for $1 \leq i \leq n$.

Set $S = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} a_1 r^{i-1}$. We will need to split into two cases.

Case 1: $r = 1$. $a_{i+1} = a_i \cdot 1 = a_i$ for $1 \leq i < n$, but then by induction $a_i = a_1$ for $1 \leq i \leq n$. So $S = \boldsymbol{na_1}$.

Case 2: $r \neq 1$. We will use a variant of Gauss' trick: we will subtract rS from S:

$$\begin{array}{rcccccccccccc}
S = & a_1 & + & a_1 r & + & a_1 r^2 & + \cdots + & a_1 r^{n-2} & + & a_1 r^{n-1} & & \\
-\,rS = & & - & a_1 r & - & a_1 r^2 & - \cdots - & a_1 r^{n-2} & - & a_1 r^{n-1} & - & a_1 r^n \\
\hline
(1-r)S = & a & + & 0 & + & 0 & + \cdots + & 0 & + & 0 & - & a_1 r^n
\end{array}$$

so $S = \dfrac{\boldsymbol{a_1(1-r^n)}}{\boldsymbol{1-r}}$.

One commonly sees geometric series written in the form $\sum_{i=0}^{n-1} ar^i$. By setting $j = i+1$, we see that $\sum_{i=0}^{n-1} ar^i = \sum_{j=1}^{n} ar^{j-1}$, so plugging in the formula derived above, we get $\sum_{i=0}^{n-1} ar^i = \frac{\boldsymbol{a(1-r^n)}}{\boldsymbol{1-r}}$.

Notice that in both examples, the derivation of the closed form involved proving something about the form of a_i via induction, and then combining the sum S with itself in a clever way. In general, if someone hands you an expression they claim to be a closed form (which may or may not be obtained by one of the above methods), you can (and probably should) verify it by induction as well.

Recurrences

When working with recursively defined sets, one can often characterize their properties in terms of recurrences.[2]

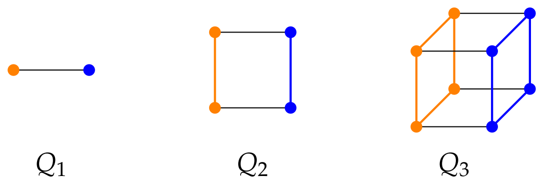
Example 3. The set of *boolean hypercubes*[3] can be constructed recursively as follows. First, the graph Q_0 consisting one vertex and no edges is a boolean hypercube.

Q_0

Now let Q_i be a boolean hypercube. We will construct a graph Q_{i+1} by taking then union of Q_i with a disjoint copy Q_i', and then adding in all edges between vertices that are copies of each other. Then Q_{i+1} is also a boolean hypercube. Below, for $0 \leq i \leq 2$, we see Q_{i+1} being constructed from two copies of Q_i, which have been colored orange and blue for visualization purposes:

[2] Next week, when we look at analyzing algorithms, we will also see that the running time of recursive functions can be analyzed in terms of recurrences.

[3] These graphs are called boolean hypercubes due to the following alternative (non-recursive) construction: Q_n is the graph whose vertices are $\{0,1\}^n$, and there is an edge between $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{y} = (y_1, \ldots, y_n)$ if and only if $\bar{x}$ and $\bar{y}$ differ on exactly one entry.



Let $VQ(i) = |V(Q_i)|$ and $EQ(i) = |E(Q_i)|$.

We know that $VQ(0) = 1$, and for $i > 0$, the vertices of Q_i are two copies of the vertices of Q_{i-1}, i.e., $VQ(i) = 2VQ(i-1)$.

Similarly, we know that $EQ(0) = 0$, and for $i > 0$ the edges of Q_i are two copies of the edges of Q_{i-1}, plus one edge for each vertex of Q_{i-1}, i.e., $EQ(i) = 2EQ(i-1) + VQ(i-1)$.

The formulas

$$VQ(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2VQ(n-1) & \text{if } n > 0 \end{cases}$$

$$EQ(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2EQ(n-1) + VQ(n-1) & \text{if } n > 0 \end{cases}$$

are examples of ***recurrences***: functions whose values are defined in terms of the same function but on smaller inputs (called *recursive calls*).[4]

[4] Recurrences where $f(n)$ is defined in terms of $f(n-1)$, called "recurrences of the first order," are often in the form $f(n) - f(n-1) = g(n)$. This form (correctly) suggests that such recurrences can be thought of as the discrete analogue to ordinary differential equations. Here $f(n) - f(n-1) = g(n)$ corresponds to the ODE $\frac{df(x)}{dx} = g(x)$.

As in the case of summations, we can ask for a ***closed form*** expression, which does not involve any recursive calls. We will also ask that the closed form not include any summations. The motivation is the same as before: we want something that can be evaluated relatively quickly, as well as give more intuition than the recursive form.

We will walk through finding closed forms for some simple recurrences. On the other hand, if someone ever hands you an expression they claim to be a closed form for some recurrence, you can (and should) verify it by (structural) induction.

Unrolling

One convenient method for finding closed forms for recurrences is called ***unrolling***. We will illustrate the steps for this method by example.

Example 4. You might have already guessed that since the number of vertices doubles each time that a closed form for $VQ(n)$ would simply be 2^n. We will see how to use the unrolling method to achieve the same result.

a) "Unroll" three or four times:

1. $VQ(n) = 2VQ(n-1)$

2. $VQ(n) = 2VQ(n-1) = 2(2VQ(n-2)) = 2^{2}VQ(n-2)$
3. $VQ(n) = 2^2VQ(n-2) = 2^2(2VQ(n-3)) = 2^{3}VQ(n-3)$

b) **Predict** the form obtained after unrolling k times:

k. $VQ(n) = 2^{k}VQ(n-k)$

c) Find the value of k that matches the base case(s):

The base case for VQ is $VQ(0) = 1$. If $n - k = 0$, then $n = k$.

d) Substitute the base case(s) and simplify:

When $k = n$, $VQ(n) = 2^{n}VQ(0) = 2^n \cdot 1 = 2^n$.

e) Sanity check on a few values, or, ideally, prove correctness by induction:[5]

Sanity check:

- Q_0: $VQ(0) = 1 = 2^0$ ✓
- Q_1: $VQ(1) = 2 = 2^1$ ✓
- Q_2: $VQ(2) = 4 = 2^2$ ✓
- Q_3: $VQ(3) = 8 = 2^3$ ✓

[5] Do not skip this step! It is very tempting to want to rest after previous four steps, but in each step there are many ways to introduce bugs into your solution.

Proof by induction:

- Base case: for $n = 0$, $VQ(0) = 1 = 2^0$.
- Inductive case: assume for $0 \le i < k$ that $VQ(i) = 2^i$. Then $VQ(k) = 2VQ(k-1) = 2 \cdot 2^{k-1} = 2^k$.

So we conclude that $\boxed{VQ(n) = 2^n}$.

Let us consider another example, which does not have to do with boolean hypercubes.

Example 5. Consider the function H defined over positive powers of two (i.e., 2^i for $i > 0$),

$$H(n) = \begin{cases} 1 & \text{if } n = 2 \\ H(\frac{n}{2}) + n & \text{otherwise} \end{cases}$$

We will compute a closed form for T by unrolling.

a) "Unroll" three or four times:

1. $H(n) = H(\frac{n}{2}) + n$
2. $H(n) = H(\frac{n}{2}) + n = (H(\frac{\frac{n}{2}}{2}) + \frac{n}{2}) + n = H(\frac{n}{2^2}) + (1 + \frac{1}{2})n$
3. $H(n) = H(\frac{n}{2^2}) + (1 + \frac{1}{2})n = (H(\frac{\frac{n}{2^2}}{2}) + \frac{n}{2^2}) + (1 + \frac{1}{2})n = H(\frac{n}{2^3}) + (1 + \frac{1}{2} + \frac{1}{4})n$

b) **Predict** the form obtained after unrolling k times:

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

c) Find the value of k that matches the base case(s):

The base case for H is $H(2) = 1$. If $\frac{n}{2^k} = 2$, then $n = 2^{k+1}$, i.e., $k = \log_2(n) - 1$.

d) Substitute the base case(s) and simplify:

When $k = \log_2(n) - 1$, $H(n) = H(2) + n(\sum_{i=0}^{\log_2(n)-2} \frac{1}{2^i}) = 1 + n\frac{1-\frac{1}{2^{\log_2 n - 1}}}{\frac{1}{2}} = 1 + 2n(1 - \frac{2}{n}) = 2n - 3$.

e) Sanity check on a few values, or, ideally, prove correctness by induction:[6]

Sanity check:

- $H(2) = 1 = 2(2) - 3$ ✓
- $H(4) = H(2) + 4 = 5 = 2(4) - 3$ ✓
- $H(8) = H(4) + 8 = 13 = 2(8) - 3$ ✓

So we conclude that $\boxed{H(n) = 2n - 3}$.

[6] Once again, do not skip this step! When writing these notes, I introduced errors more than once, and they were caught precisely because of this step.

Recursion Trees

For recurrences of the form

$$T(n) = \begin{cases} g(n) & n \text{ is a base case} \\ aT(\frac{n}{b}) + f(n) & \text{otherwise} \end{cases}$$

where a, b are integers and (for simplicity) n is a power of b, another useful method for finding closed forms is the ***recursion tree method***.

In this method, we will say that $T(n)$ consists of a ***recursive calls*** to $T(\frac{n}{b})$ and $f(n)$ ***extra work***. As for the recursion tree itself, the root has value $f(n)$ and a children, each of which is the root of a (recursively constructed) recursion tree for $T(\frac{n}{b})$. When $T(\frac{n}{b})$ hits one of the base cases, we get a leaf whose value is simply $g(\frac{n}{b})$.

As before, we will illustrate this method by example.

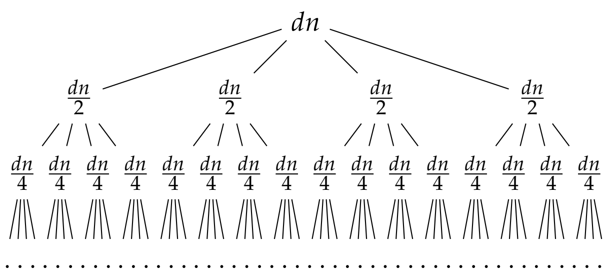
Example 6. Consider the following recurrence, defined for powers of two that are at least four:

$$S(n) = \begin{cases} c & \text{if } n = 4 \\ 4S(\frac{n}{2}) + dn & \text{otherwise} \end{cases}$$

We will compute a closed form for A by recursion trees.

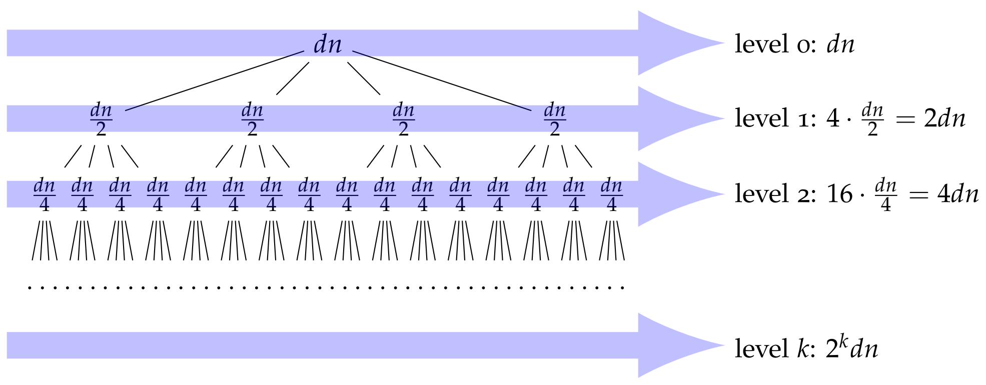
a) Draw the first few layers of the recursion tree:

For S, $a = 4$, $b = 2$, and $f(n) = dn$, so the first few levels of the recursion tree are as follows:



b) **Predict** the total amount of extra work at each (internal) level:

In the recursion tree for S, the first few levels sum to dn, $2dn$, and $4dn$. We will predict that at level k, the extra works sums to $2^k dn$.


level 0: dn
level 1: $4 \cdot \frac{dn}{2} = 2dn$
level 2: $16 \cdot \frac{dn}{4} = 4dn$
level k: $2^k dn$

c) Compute the number of internal levels as well as the number of leaves:

For S, a vertex is at level k if it corresponds to a call to $S(\frac{n}{2^k})$, and a vertex is a leaf if it corresponds to $S(4)$. Thus the leaves are at level h where $\frac{n}{2^h} = 4$, i.e., $h = \log_2(n) - 2$. This happens to be the height of the tree, and all leaves are at level $\log_2(n) - 2$. The internal levels are thus 0 through $\log_2(n) - 3$.

We can see that the number of vertices at level k is 4^k, so the number of leaves is $4^{\log_2(n)-2}$.

d) Add up the total work at each internal level, plus the work at the leaves:

$$S(n) = \overbrace{\sum_{k=0}^{\log_2(n)-3} 2^k dn}^{\text{internal work}} + \overbrace{4^{\log_2(n)-2} c}^{\text{leaf work}}$$

e) Simplify any sums and logarithms to obtain a closed form:

Using the formula for geometric series

$$\begin{aligned}\sum_{k=0}^{\log_2(n)-3} 2^k dn &= dn \sum_{k=0}^{\log_2(n)-3} 2^k \\ &= dn \cdot \frac{2^{\log_2(n)-2}-1}{2-1} \\ &= dn \cdot \left(2^{\log_2(n)-2}-1\right).\end{aligned}$$

Then using the exponent rule $a^{b+c} = a^b a^c$ and logarithm rule $a^{\log_a(b)} = b$, we can simplify

$$2^{\log_2(n)-2} = \frac{2^{\log_2(n)}}{4} = \frac{n}{4}.$$

Next, using the exponent rule $a^{b+c} = a^b a^c$ and logarithm rule $a^{\log_b(c)} = c^{\log_b(a)}$, we can simplify

$$4^{\log_2(n)-2} = \frac{4^{\log_2(n)}}{16} = \frac{n^{\log_2 4}}{16} = \frac{n^2}{16}.$$

Putting everything together, $S(n) = dn(\frac{n}{4}-1) + \frac{cn^2}{16}$.

f) Sanity check on a few values, or, ideally, prove correctness by induction:

Sanity check:

- $S(4) = c = 4d(\frac{4}{4}-1) + c \cdot \frac{4^2}{16}$ ✓
- $S(8) = 4S(4) + 8d = 8d + 4c = 8d(\frac{8}{4}-1) + c \cdot \frac{8^2}{16}$ ✓
- $S(16) = 4S(8) + 16d = 48d + 16c = 16d(\frac{16}{4}-1) + c \cdot \frac{16^2}{16}$ ✓

In summary, $\boxed{S(n) = dn\left(\frac{n}{4}-1\right) + \frac{cn^2}{16}}$.