CSCI 2200

Foundations of Computer Science

Lecture 9: Sums and Asymptotics

CSCI 2200

Foundations of Computer Science

Lecture 9: Sums and Asymptotics

"Katamari on the Rocks" performed by Masayuki Tanaka and Tomomi Suzuki, composed by Yuu Miyake

Today's tasks

- HW questions
- Methods for dealing with summations
- Asymptotic notation: big-O and its cousins

- Shameless plug: My friend P.D. Magnus recent published a paper titled On Trusting Chatbots. Worth reading.

An example problem: Subsequence sum

- Given a list of integers...

```
1 -1 -1  2  3  4 -1 -1  2  3 -4  1  2 -1 -2  1
```

- ... find the run of *consecutive* values that produces the largest sum.
- How would you accomplish this?

```
max = -sys.maxsize - 1
for i in range(N):
   for j in range(i+1,N):
      total = sum(A[i:j])
      if total > max:
         max = total
```

Calculating runtime

```
max = -sys.maxsize - 1
for i in range(n):
    for j in range(i+1,n):
        total = sum(A[i:j])
        if total > max:
            max = total
```

- How long does this take to run?
- Can we count the total operations?

$$2 + \sum_{i=1}^{n}\left[2 + \sum_{j=i}^{n}\left(5 + \sum_{k=i}^{j} 2\right)\right]$$

- Can we do better?

Other possibilities

- Here are the # of operations required for a variety of potential algorithms:
  - $T_1(n) = 2 + \sum_{i=1}^{n}\left[2 + \sum_{j=i}^{n}\left(5 + \sum_{k=i}^{j} 2\right)\right]$ (brute force)
  - $T_2(n) = 2 + \sum_{i=1}^{n}\left(3 + \sum_{j=i}^{n} 6\right)$ (one fewer *for* loop)
  - $T_3(n) = \begin{cases} 3 & n = 1 \\ 2T_3\left(\frac{1}{2}n\right) + 6n + 9 & n > 1 \text{ and even} \\ T_3\left(\frac{1}{2}(n+1)\right) + T_3\left(\frac{1}{2}(n-1)\right) + 6n + 9 & n > 1 \text{ and odd} \end{cases}$
  - $T_4(n) = 5 + \sum_{i=1}^{n} 10$ (two fewer *for* loops!)
  - Which is best?

Small cases

- We can plug in some values for n (e.g. 1, 5, 10, 20):
  - $T_1(1) = 11; T_1(5) = 157; T_1(10) = 737; T_1(20) = 4172$
  - $T_2(1) = 11; T_2(5) = 107; T_2(10) = 362; T_2(20) = 1322$
  - $T_3(1) = 3; T_3(5) = 123; T_3(10) = 315; T_3(20) = 759$
  - $T_4(1) = 15; T_4(5) = 55; T_4(10) = 105; T_4(20) = 205$
- The fourth algorithm looks best… on these small cases.
- Rarely do real programs run on such small datasets.
- We need techniques to (a) write these runtimes without summations and (b) quickly compare functions with LARGE inputs.

Dealing with summations

Rule 1: The Constant Rule

- $\sum_{i=1}^{5} 7 = 7 + 7 + 7 + 7 + 7 = 7(5)$
- $\sum_{i=1}^{5} k = k + k + k + k + k = k(5)$
- $\sum_{i=1}^{n} k = k + \cdots + k = kn$

- Rule: Multiplicative constants can be pulled out of the sum.
  - $\sum_{i=1}^{n} k(n^2 + 3n + 8) = k\sum_{i=1}^{n}(n^2 + 3n + 8)$
- But! Please notice:
  - $\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = \frac{1}{2}(5)(5 + 1)$
  - If it depends on the *index of summation*, it's not constant!

Rule 2: The Addition Rule

- $\sum_{i=1}^{5}(i + i^2) = (1 + 1^2) + (2 + 2^2) + \cdots + (5 + 5^2)$
- $= (1 + 2 + 3 + 4 + 5) + (1^2 + 2^2 + 3^2 + 4^2 + 5^2)$
- $= \sum_{i=1}^{5} i + \sum_{i=1}^{5} i^2$

- Rule: If there is an addition inside the summation, we can break this into two separate summations. Also, if there are two or more summations *with the same index & bounds*, then we can combine them into a single one.

Crib sheet material: Common summations

- $\sum_{i=k}^{n} 1 = n + 1 - k$
- $\sum_{i=1}^{n} f(x) = nf(x)$ *NOTE:* $f(x)$ must <u>not</u> contain i
- $\sum_{i=1}^{n} i = (1/2)(n)(n+1)$
- $\sum_{i=1}^{n} i^2 = (1/6)(n)(n+1)(2n+1)$
- $\sum_{i=1}^{n} i^3 = (1/4)(n^2)(n+1)^2$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ In binary, 111…111 + 1 = 1000…000
- $\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$ Each new term gets you halfway to 2.
- $\sum_{i=1}^{n} \log i = \log n!$
- $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$ *Note:* $r \neq 1$; This is called the *<u>geometric series</u>*.

Example summation breakdown

- $\sum_{i=1}^{n}(1 + 2i + 2^{i+2})$
- $\sum_{i=1}^{n} 1 + 2\sum_{i=1}^{n} i + \sum_{i=1}^{n} 2^{i+2}$
- $n + 2(\frac{1}{2})(n)(n+1) + \sum_{i=1}^{n} 2^2 2^i$
- $n^2 + 2n + 2^2 \sum_{i=1}^{n} 2^i$
- $n^2 + 2n + 2^2(2^{n+1} - 1 - 1)$
- $2^{n+3} + n^2 + 2n - 8$
- Expressions without summations are referred to as *closed-form* expressions – they allow us to make direct calculations.

You try it!

- $\sum_{i=1}^{n}(5i + 2n)$
- $\sum_{i=1}^{n} 5i + \sum_{i=1}^{n} 2n$
- $5\sum_{i=1}^{n} i + 2\sum_{i=1}^{n} n$
- $5\left(\frac{1}{2}\right)(n)(n + 1) + 2(n)(n)$
- $\frac{9}{2}n^2 + \frac{5}{2}n$

Rule 3: The Nested Sum Rule

- To compute a nested summation, start with the innermost sum and work outward.
  - $\sum_{i=1}^{5} \sum_{j=1}^{5} 1 = \sum_{i=1}^{5} 5 = 5(5) = 25$
  - $\sum_{i=1}^{5} \sum_{j=1}^{i} 1 = \ \dots ?$
  - $\sum_{i=1}^{5} \sum_{j=1}^{i} 1 = \ \sum_{i=1}^{5} i = \left(\frac{1}{2}\right)(5)(5+1) = 15$

A larger example

- Remember our runtimes?
  - $T_2(n) = 2 + \sum_{i=1}^{n}\left(3 + \sum_{j=i}^{n} 6\right)$
  - $T_2(n) = 2 + \sum_{i=1}^{n} 3 + \sum_{i=1}^{n}\sum_{j=i}^{n} 6$
  - $T_2(n) = 2 + 3n + \sum_{i=1}^{n} 6 \sum_{j=i}^{n} 1$
  - $T_2(n) = 2 + 3n + 6\sum_{i=1}^{n}(n + 1 - i)$
  - $T_2(n) = 2 + 3n + 6\left(n^2 + n - \left(\frac{1}{2}\right)(n)(n+1)\right)$
  - $T_2(n) = 2 + 3n + 6\left(\frac{1}{2}n^2 + \frac{1}{2}n\right)$
  - $T_2(n) = 3n^2 + 6n + 2$

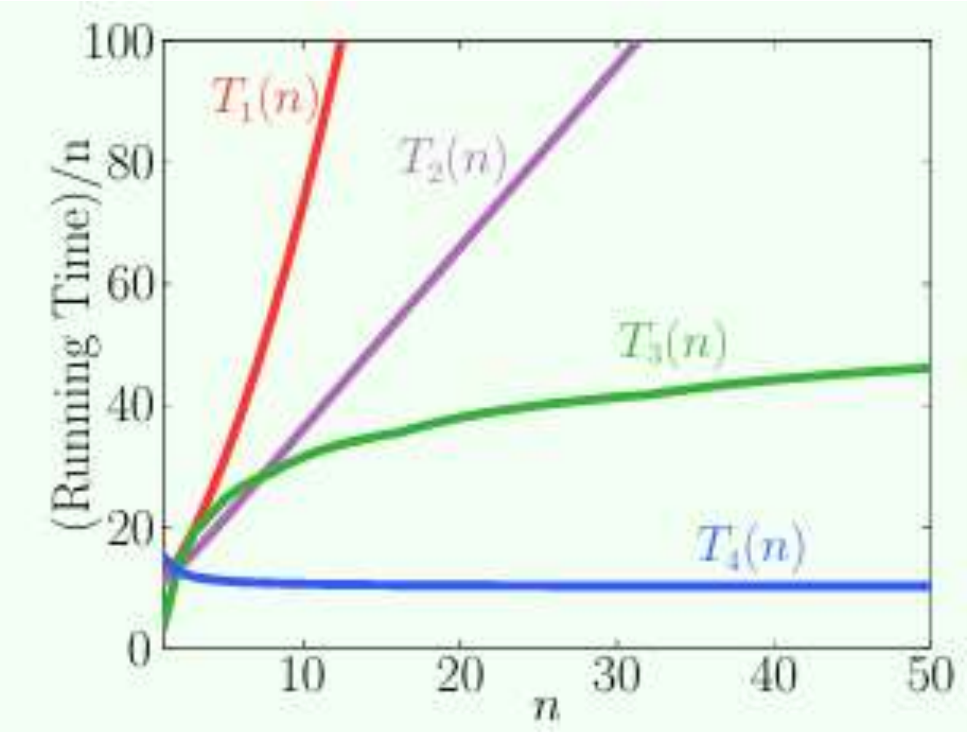
You try it!

- $\sum_{i=1}^{n} \sum_{j=1}^{i} ij$
- $\sum_{i=1}^{n} i \sum_{j=1}^{i} j$ by the Constant Rule! i is unaffected by j
- $\sum_{i=1}^{n} i \left(\frac{1}{2}\right)(i)(i+1)$
- $\sum_{i=1}^{n} \frac{1}{2}(i^3 + i^2)$
- $\frac{1}{2}(\sum_{i=1}^{n} i^3 + \sum_{i=1}^{n} i^2)$
- $\frac{1}{2}\left(\frac{1}{4}n^2(n+1)^2 + \frac{1}{6}n(n+1)(2n+1)\right)$
- $\frac{1}{8}n^4 + \frac{5}{12}n^3 + \frac{3}{8}n^2 + \frac{1}{12}n$

Asymptotic analysis (i.e. big-O notation)

Back to our runtimes

- $T_1(n) = \frac{1}{3}n^3 + \frac{7}{2}n^2 + \frac{31}{6}n + 2$
- $T_2(n) = 3n^2 + 6n + 2$
- $T_3(n) \leq 12n(\log n + 3) - 9$
- $T_4(n) = 10n + 5$
  - Why is $T_3 \leq$ instead of =? Well, it wasn't a sum… Recursions have their own rules, which are mostly not part of FOCS.
- How do these compare for large n?



The big BIG idea

- Big Data is all the rage these days. So our production algorithms need to finish in reasonable time for really, really, *really* large values of n.
  - Terabyte $\approx 2\hat{}40$ bytes, or around 1,000,000,000,000 bytes
- For "big enough" n, coefficients are nearly irrelevant – the only things that tend to matter are *exponents*.
  - n to a higher power will always be worse than n to a lower power, no matter what constants we multiply by. $50000n^2$ is less runtime than n^3 eventually.
  - And n in the exponent itself is just … AWFUL.
- We would like a tool to express this particular concept of "better".

Asymptotic analysis

- Asymptotic analysis looks at the behavior of functions as their input $n \to \infty$.
- The most common tool for this is "big-O notation". It expresses the idea that a runtime is **no worse than** some function in the long run.
- Formally, $T(n) \in O\big(f(n)\big)$ means that:
$$\exists C > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, T(n) \leq C \cdot f(n)$$
  - "Once n is big enough (i.e. $\geq n_0$), $T(n)$ is no worse than $f(n)$ times some fixed constant C."

Practical application

- What this means is that, when performing asymptotic analysis:
  - We generally only worry about the worst-case input.
  - We can ignore constant coefficients. $60n^2 \in O(n^2)$
    - Side note: The bounds for O do not have to be tight. $60n^2 \in O(n^{12})$
  - We can throw out any lower-order terms. $5n^3 + 3n^2 + 1000 \in O(n^3)$
- When looking at sums, every summation (usually) adds a factor of n to whatever is inside the summation.
  - $\sum_{i=1}^{n} \sum_{j=1}^{i} ij \in O(n^4)$ since the inside part (ij) is effectively quadratic.

The big-O menagerie

- The common classes of functions we work with are:
  - $O(1)$ – constant (*array access*)
  - $O(\log n)$ - logarithmic (*binary search in a sorted array*)
  - $O(n)$ – linear (*search in an unsorted array*)
  - $O(n \log n)$ - loglinear (*good sorting algorithms like mergesort*)
  - $O(n^2)$ - quadratic (*poor sorting algorithms like bubble sort*)
  - $O(n^c)$ - higher-order polynomials (*some parsing algorithms*)
  - $O(n^{\log n})$ – quasipolynomial (*old primality test*)
  - $O(2^n)$ - exponential (*Traveling Salesman Problem*)
  - $O(n!), O(n^n), O\left(n^{2^n}\right)$, and even worse – Don't go there.

Thought for the day

- "$O(n^2)$ is the sweet spot of badly scaling algorithms: fast enough to make it into production, but slow enough to make things fall down once it gets there." --Bruce Dawson
- This blog post details a *lovely* little bug that shipped with Windows 11, in which one of its UI threads would sometimes hang...
  - ... because it used a quadratic-time algorithm to place desktop icons nicely onto a grid ...
  - ... even if those icons weren't being displayed right now!

Siblings of big-O…

- $f(n) \in O(g(n))$ is a "less than or equal to" relation.
- Unsurprisingly, there is also a "greater than or equal to" relation: big-Ω
  - $T(n) \in \Omega(f(n))$ means that $$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0, T(n) \geq c \cdot f(n)$$
  - If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$.
- What if $f(n) \in O(g(n))$ <u>and</u> $f(n) \in \Omega(g(n))$? Big-Θ
  - We say that $g(n)$ is an *<u>asymptotically tight</u>* bound on $f(n)$ – they are equals of a sort. We write $f(n) \in \Theta(g(n))$, which means $\exists c, C > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, c \cdot g(n) \leq f(n) \leq C \cdot g(n)$.

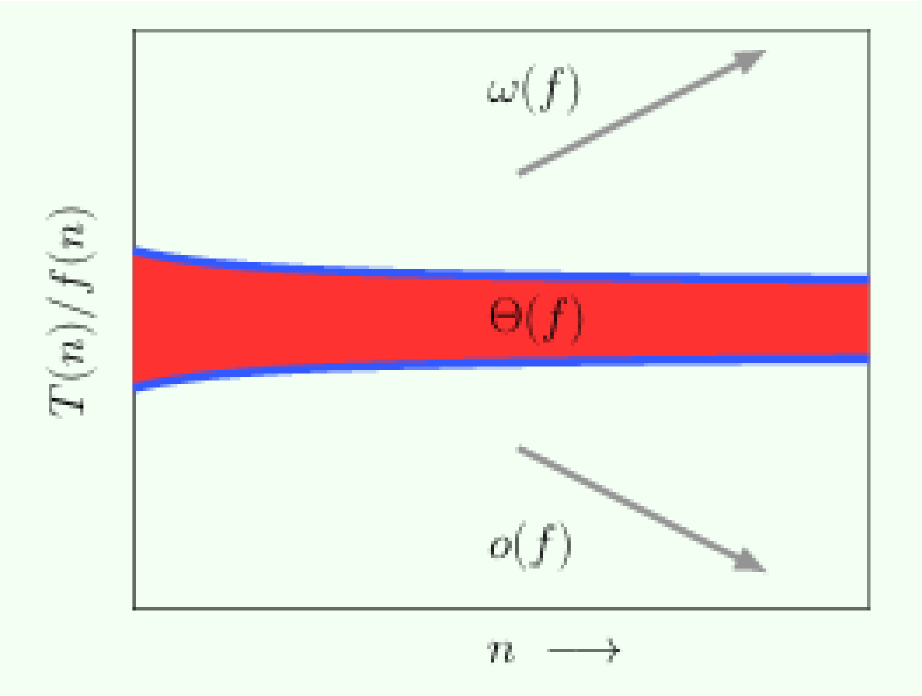
… and their little cousins

- We also have two relations that are comparable to the strict inequalities > and <, but they are not defined formally the same way.
- $f(n) \in o\big(g(n)\big)$ means that $f(n)$ is of a strictly smaller order than $g(n)$ – that is $\forall c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) < c \cdot g(n)$
  - Note the order of the quantifiers here: First we pick a constant, then define a "big enough" value. The other way wouldn't work.
  - But, importantly, for <u>any</u> constant, there's always "big enough."
- The reverse relation is $f(n) \in \omega\big(g(n)\big)$, which means $f(n)$ is of strictly greater order than $g(n)$.

Using limits to express the same ideas

$$\lim_{n\to\infty}\frac{T(n)}{f(n)} = \cdots$$

- ... 0, then $T(n) \in o(f(n))$.
- ... any $c > 0$, then $T(n) \in \Theta(f(n))$.
- ... ∞, then $T(n) \in \omega(f(n))$.



- In practice, we nearly always just use big-O.

Questions?