.

Motivation; Introduction and Mathematical Foundations

Introduction and Motivation

To introduce you to what you will learn in this course, we'll look at a couple of simple puzzles.

⋆ **Puzzle 1:** You have a seven link chain. You want to cut this chain such that you can give a person one link one night, then two links the next night, and so on up to seven links on the seventh night. Your goal is to minimize the number of cuts.

⋆ **Puzzle 2:** You are given 12 golf balls, one of which is heavier than the rest, and a balancing scale. You want to find the heavier ball, using as few weighings as possible.

In this course, for several problem domains, we are given

- A **computational problem** to solve (i.e. given an input produce an output).
- A "**model of computation**" i.e. a set of allowable operations.
- A **resource to measure**.

Our goal is to design an **algorithm** that solves the problem using the model, while minimizing the resources used.

⋆ **Puzzle 1 (seven links problem):**

- Input:
- Output:
- Allowable operation:
- Measurable resource:

The situation is somewhat different for the problems we will study, because a computational problem is a **set of instances** of varying sizes. As a result, the algorithm must be designed to handle all possible input sizes. For this reason, we typically measure the resources used as a function of the size of the instance. (Think about the golf ball problem with n balls.) Examples of the types of problems to be studied during the semester are:

1. **Problems on Sorting and Searching.**

   ⋆ **Sorting problem:**

   - Input: Array of n integers.
   - Output:
   - Allowable operation:
   - Measurable resource:

2. **Problems on Graphs.** Graphs can be used to model all kinds of interesting problems. **Think of the World Wide Web as a graph and all sorts of problems arise**. e.g., how to compute the shortest path between a given pair of nodes, or how to compute the likelihood that a node (webpage) will be visited.

3. **Problems on Strings.** An example of a problem in this area that you should be familiar with from using editors is: given a pattern and a text file, find all instances of the pattern in the text file. Text compression is another example that is becoming increasingly important. Algorithms for comparing strings have myriad applications in computational biology, such as in sequencing the human genome.

4. **Problems in Algebra and Number Theory.**

   Number theory algorithms are the basis for public key cryptosystems.

For each problem domain, we will be interested in the following:

1. **Algorithm design.** The goal is to design an algorithm that minimizes some resource, as a function of input size. Coming up with good algorithms for these problem domains can be tricky. But there is a handful of design techniques that we will learn, which are useful for a wide variety of problems.

2. **Reasoning about the correctness of algorithms.** Once you have a promising-looking algorithm, how do you know if it is correct? We'll look at ways of testing and verifying that algorithms are correct.

3. **Algorithm analysis.** Once you believe your algorithm is correct, how do you measure the resources used? Tools that will help us with this include solving **recurring relations** and finding **closed-form expressions** for summations. We will also be interested in comparing

algorithms, in terms of the resources used, to see which is better. **O-notation** will be very helpful for this purpose. We will usually be concerned with two types of analysis:

- **Worst case analysis.** For example, suppose we are analyzing a sorting algorithm and are counting the number of swaps done. It may be the case that the number of comparisons done by the algorithm on an array of n elements depends on the relative order of the elements initially in the array. When doing worst case analysis, we want to know what is the *maximum* number of comparisons that the algorithm does, over all possible input arrays with n elements.
- **Average case analysis** is often concerned with estimating the *expected* number of comparisons (as a function of n), averaged over all possible inputs of size n.

4. **Lower bounds.** Finally, how do you know that your algorithm is the best possible, in terms of minimizing the required resources? For some problems, we will develop lower bounds towards this end.

Summations

The next lectures will cover mathematical background that is used in analyzing algorithms, starting with summations.

⋆ **Example 1:** $\sum_{i=1}^{n} i$**.**

Take the following simple code segment:

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow i$ **to** n **do**

    [code]

How many times is [code] executed?

We are interested in finding a **closed form expression** for this summation. That is, we'd like to express the summation as a function of n. With the closed form expression in hand, it is easier to compare the number of times [code] is executed by this algorithm with other algorithms, for example.

You may have seen this summation before and recall that the solution is

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

In general, if you have a guess as to what the solution of a summation is, you may be able to use **proof by induction** to *prove* that your answer is correct. It is handy to be able to use proof by induction in this way, since often you vaguely remember the closed form expression for a summation but may not be sure you remember it correctly.

⋆ **Claim:** $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

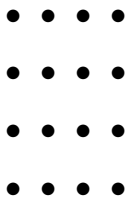
Proof:

Basis: First we prove that the claim is true for some n, say $n = 1$:

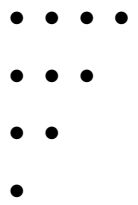
Induction Step: Now we assume that the claim is true for n, and prove that it is true for $n + 1$:

Therefore, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all n. ■

If you are not already familiar with the summation of example 1, you can derive it using a geometrical approach. If the code segment: (**for** $j \leftarrow i$ **to** n **do**) began at 1, instead of at i, [code] would be executed n^2 times. The graphical representation would look like a square of dots.



But since the **for** loop starts at i, instead of 1, the actual graphical representation would be a triangle of dots (basically, 1/2 of the square including the diagonal).



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

Still another approach is the method that Gauss employed as a student: The story goes that he was given the task of summing up all the numbers from 1 to 100. Rather than tediously adding each number in sequence, Gauss observed that if the numbers were paired as follows:

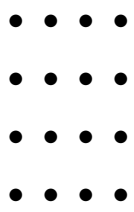
1	2	3	4	5	...	48	49	50
100	99	98	97	96	...	53	52	51

the sum of each numbered pair equaled 101, and there were 50 pairs of numbers. He then proceeded to calculate $101 \times 50 = 5050$. More generally with n numbers (n even) the sum is

$$(n+1) \cdot \frac{n}{2} = \frac{n(n+1)}{2}.$$

⋆ **Example 2:**

How many subsquares of size 1×1, 2×2, ... are there in an $n \times n$ square?



Can you find a summation whose value is the desired number of subsquares? Can you find a closed form expression for this summation? One way to approach this problem is to consider small values of n.

- For $n = 2$: 4 (1×1) and 1 (2×2) squares.
- For $n = 3$:
- For $n = 4$:
- It is starting to look like the number of squares in an $n \times n$ square is

Here is another approach to answering this question. Each $k \times k$ square to be counted has a unique dot of the grid as its top left corner. How many dots from the first row can be the top left dot of some $k \times k$ square?

⋆ Consider the case $k = 2$ first.

An extension of this argument shows that there are $(n-k+1)^2$ $k \times k$ squares. To sum up all the squares, we can let k run from 1 to n (the smallest-sized square to the largest-sized square, respectively). Hence, the total number of squares is

⋆

We need to find a "closed form" expression for $\sum_{k=1}^{n} k^2$. Since $\sum_{k=1}^{n} k$ is a quadratic polynomial in n, a reasonable guess is that $\sum_{k=1}^{n} k^2$ is a cubic polynomial, i.e. of the form

$$An^3 + Bn^2 + Cn + D,$$

where A, B, C and D are constants to be determined. If we can generate four independent equations containing our four unknowns, we can solve these to compute A, B, C and D. The four independent equations can be generated by letting n in the summation take on four different values, for simplicity, say 1, 2, 3 and 4. This gives us

⋆

$$n = 1: \quad \sum_{k=1}^{1} k^2 = 1 =$$

$$n = 2: \quad \sum_{k=1}^{2} k^2 = 1^2 + 2^2 =$$

$$n = 3: \quad \sum_{k=1}^{3} k^2 = 1^2 + 2^2 + 3^2 = 14 =$$

$$n = 4: \quad \sum_{k=1}^{4} k^2 =$$

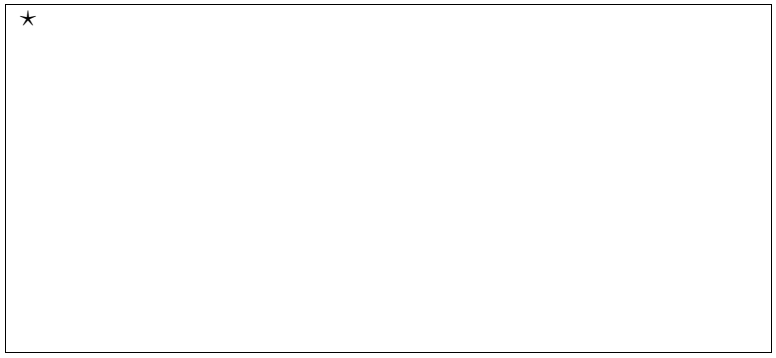
Solving the four independent equations simultaneously yields $A = 1/3$, $B = 1/2$, $C = 1/6$, $D = 0$. Therefore

$$\begin{aligned} \sum_{k=1}^{n} k^2 &= \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n + 0 \\ &= \cdots \text{(algebraic manipulation)} \cdots \\ &= \frac{n(n+1)(2n+1)}{6}. \end{aligned}$$

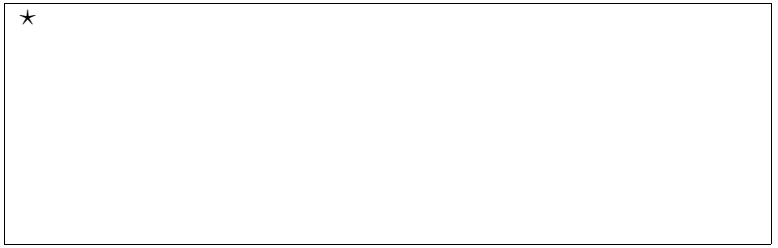
So, if our guess (that $\sum_{k=1}^{n} k^2$ is a cubic polynomial) is correct, we have the desired closed form expression. *Since the answer is based on a guess, though, we still haven't proved that* $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$. Luckily, we can now use proof by induction (as in the previous summation) to verify that this is correct.

Lower and Upper Bounds

We next show simple upper and lower bounds on $\sum_{k=1}^{n} k^2$. For an upper bound:



and for a lower bound:



Therefore, by just calculating very rough bounds, we can see that the n^3 term in our guess for the polynomial falls within these bounds. If our main interest is not in exact numbers, but rather in finding the order of the growth rate of our summations, these bounds show us that the summation $\sum_{k=1}^{n} k^2$ has a growth rate proportional to the function n^3. (More on this in the next lecture.) Throughout this course, we will be making these "back of the envelope" estimations.