Week 6 - Wednesday

COMP 4500

Last time

- What did we talk about last time?
- Finished Exam 1 post mortem
- Data compression example
- Mergesort

Questions?

Assignment 3

Logical warmup

- On the planet Og, there are green people and red people
- Likewise, there are northerners and southerners
- Green northerners tell the truth
- Red northerners lie
- Green southerners lie
- Red southerners tell the truth
- Consider the following statements by two natives named Ork and Bork:
  - **Ork:** Bork is from the north
  - **Bork:** Ork is from the south
  - **Ork:** Bork is red
  - **Bork:** Ork is green
- What are the colors and origins of Ork and Bork?

Divide and Conquer

Recursive running time

- If we can, we want to turn the recursive version of ***T***(***n***) into an explicit (non-recursive) Big Oh bound
- Before we do, note that we could similarly have written:

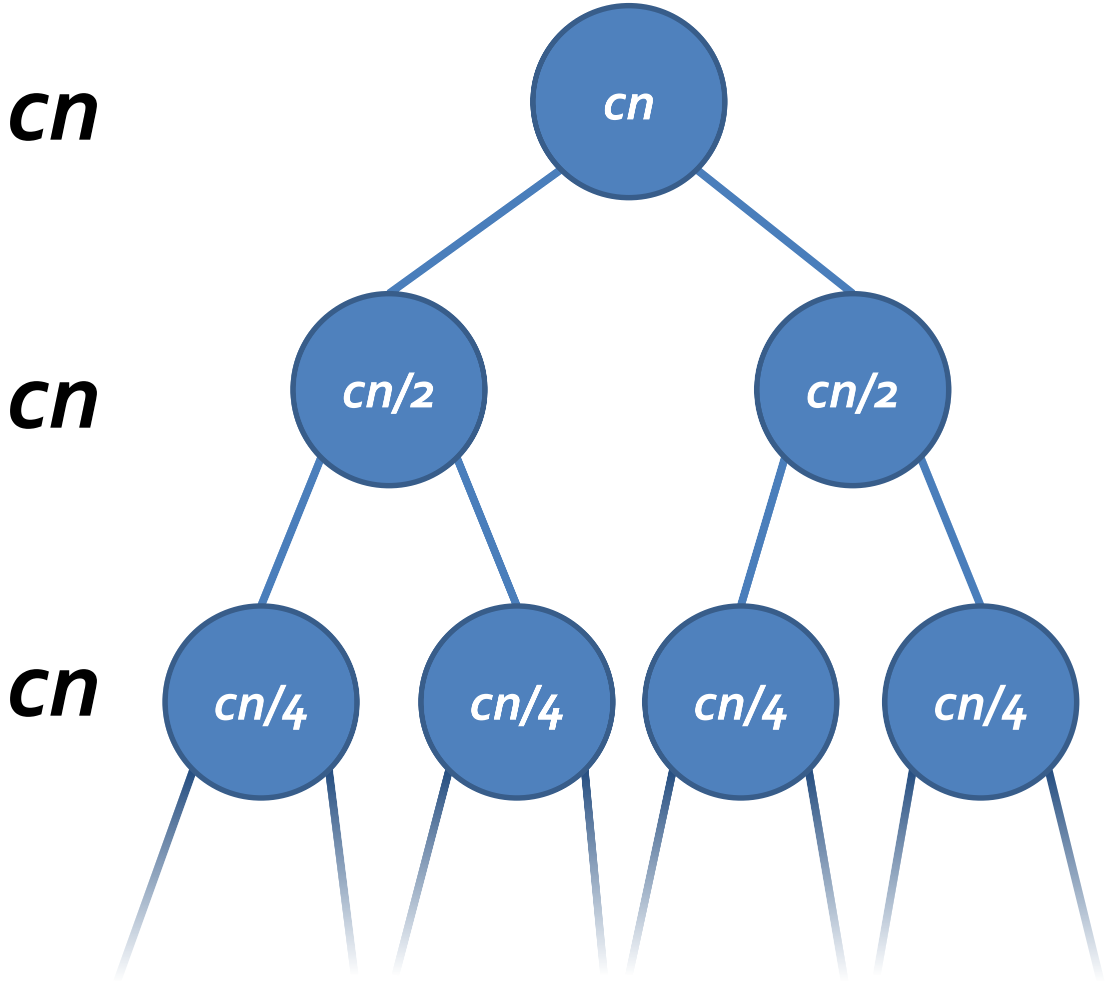
$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$$

- Also, we can't guarantee that ***n*** is even
- A more accurate statement would be

$$T(n) \leq T\left(\left\lceil\frac{n}{2}\right\rceil\right) + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + cn$$

- Usually, we ignore that issue and assume that ***n*** is a power of 2, evenly divisible forever

Intuition about mergesort recursion



- Each time, the recursion cuts the work in half while doubling the number of problems
  - The total work at each level is thus always ***cn***
- To go from ***n*** to 2, we have to cut the size in half $(\log_2 n) - 1$ times

Checking a solution

- We know that there's ***cn*** work at each level and approximately $\log_2$ ***n*** levels
- If we think that the running time O(***n*** log ***n***), we can guess that ***T***(***n***) ≤ ***cn*** $\log_2$ ***n*** and substitute that in for ***T***(***n***/2)

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

$$\leq 2c\left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right) + cn$$

$$= cn\,(\log_2 n - 1) + cn$$

$$= cn\ \log_2 n - cn + cn$$

$$= cn\log_2 n$$

Divide and conquer

- **Divide and conquer** algorithms are ones in which we divide a problem into parts and recursively solve each part
- Then, we do some work to combine the solutions to each part into a final solution
- Divide and conquer algorithms are often simple
- However, their running time can be challenging to compute because recursion is involved

Recursively defined sequences

- Defining a sequence recursively as with Mergesort is called a **recurrence relation**
- The **initial conditions** give the starting point
- Example:
  - Initial conditions
    - $T(0) = 1$
    - $T(1) = 2$
  - Recurrence relation
    - $T(k) = T(k-1) + kT(k-2) + 1$, for all integers $k \geq 2$
  - Find $T(2)$, $T(3)$, and $T(4)$

Writing recurrence relations in multiple ways

- Consider the following recurrence relation:
  - $T(k) = 3T(k-1) - 1$, for all integers $k \geq 1$
- Now consider this one:
  - $T(k+1) = 3T(k) - 1$, for all integers $k \geq 0$
- Both recurrence relations have the same meaning

Differences in initial conditions

- Even if the recurrence relations are equivalent, different initial conditions can cause a different sequence
- Example:
  - $T(k) = 3\,T(k-1)$, for all integers $k \geq 2$
  - $T(1) = 2$
  - $S(k) = 3S(k-1)$, for all integers $k \geq 2$
  - $S(1) = 1$
  - Find $T(1)$, $T(2)$, and $T(3)$
  - Find $S(1)$, $S(2)$, and $S(3)$

Compound interest

- Interest is compounded based on some period of time
- We can define the value recursively
- Let i is the **annual percentage rate** (APR) of interest
- Let m be the number of times per year the interest is compounded
- Thus, the total value of the investment at the k^{th} period is
  - $P(k) = P(k-1) + P(k-1)(i/m), k \geq 1$
  - $P(0)$ = initial principle

Solving Recurrence Relations

Recursion

- ... is confusing
- We don't naturally think recursively (but perhaps you can raise your children to think that way?)
- With an interest rate of $\boldsymbol{i}$, a principle of $\boldsymbol{P}(0)$, and $\boldsymbol{m}$ periods per year, the investment will yield $\boldsymbol{P}(0)(\boldsymbol{i}/\boldsymbol{m} + 1)^{k}$ after $\boldsymbol{k}$ periods

Finding explicit formulas by iteration

- We want to be able to turn recurrence relations into explicit formulas whenever possible
- Often, the simplest way is to find these formulas by **iteration**
- The technique of iteration relies on writing out many expansions of the recursive sequence and looking for patterns
- That's it

Iteration example

- Find a pattern for the following recurrence relation:
  - $T(k) = T(k-1) + 2$
  - $T(0) = 1$
- Start at the first term
- Write the next below
- Do not combine like terms!
- Leave everything in expanded form until patterns emerge

Arithmetic sequence

- In principle, we should use mathematical induction to prove that the explicit formula we guess actually holds
- The previous example (odd integers) shows a simple example of an arithmetic sequence
- These are recurrences of the form:
  - $T(k) = T(k-1) + d$, for integers $k \geq 1$
- Note that these recurrences are always equivalent to
  - $T(n) = T(0) + dn$, for all integers $n \geq 0$

Geometric sequence

- Find a pattern for the following recurrence relation:
  - $T(k) = rT(k-1)$, $k \geq 1$
  - $T(0) = a$
- Again, start at the first term
- Write the next below
- Do not combine like terms!
- Leave everything in expanded form until patterns emerge

Geometric sequence

- It appears that any geometric sequence with the following form
  - $\boldsymbol{T}(\boldsymbol{k}) = \boldsymbol{rT}(\boldsymbol{k}-1)$, $\boldsymbol{k} \geq 1$
- is equivalent to
  - $\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{T}(0)\boldsymbol{r}^{\boldsymbol{n}}$, for all integers $\boldsymbol{n} \geq 0$
- This result applies directly to compound interest calculation

Employing outside formulas

- Intelligent pattern matching gets you a long way
- However, it is sometimes necessary to substitute in some known formula to simplify a series of terms
- Recall
  - Geometric series: $1 + \boldsymbol{r} + \boldsymbol{r}^2 + \ldots + \boldsymbol{r}^{\boldsymbol{n}} = (\boldsymbol{r}^{\boldsymbol{n}+1} - 1)/(\boldsymbol{r} - 1)$
  - Arithmetic series: $1 + 2 + 3 + \ldots + \boldsymbol{n} = \boldsymbol{n}(\boldsymbol{n} + 1)/2$

How many edges are in a complete graph?

- In a complete graph, every node is connected to every other node
- If we want to make a complete graph with $\boldsymbol{k}$ nodes, we can take a complete graph with $\boldsymbol{k} - 1$ nodes, add a new node, and add $\boldsymbol{k} - 1$ edges (so that all the old nodes are connected to the new node)
- Recursively, this means that the number of edges in a complete graph is
  - $\boldsymbol{S}(\boldsymbol{k}) = \boldsymbol{S}(\boldsymbol{k}-1) + (\boldsymbol{k} - 1)$, $\boldsymbol{k} \geq 2$
  - $\boldsymbol{S}(1) = 0$ (no edges in a graph with a single node)
- Use iteration to solve this recurrence relation

How long does binary search take?

- We can model the running time for binary search as a recurrence relation
  - $T(n) = T(n/2) + c$, $k \geq 2$
  - $T(1) = c$
- Use iteration to solve this recurrence relation
- Instead of plugging in values 1, 2, 3,… , try powers of two: 1, 2, 4, 8,…

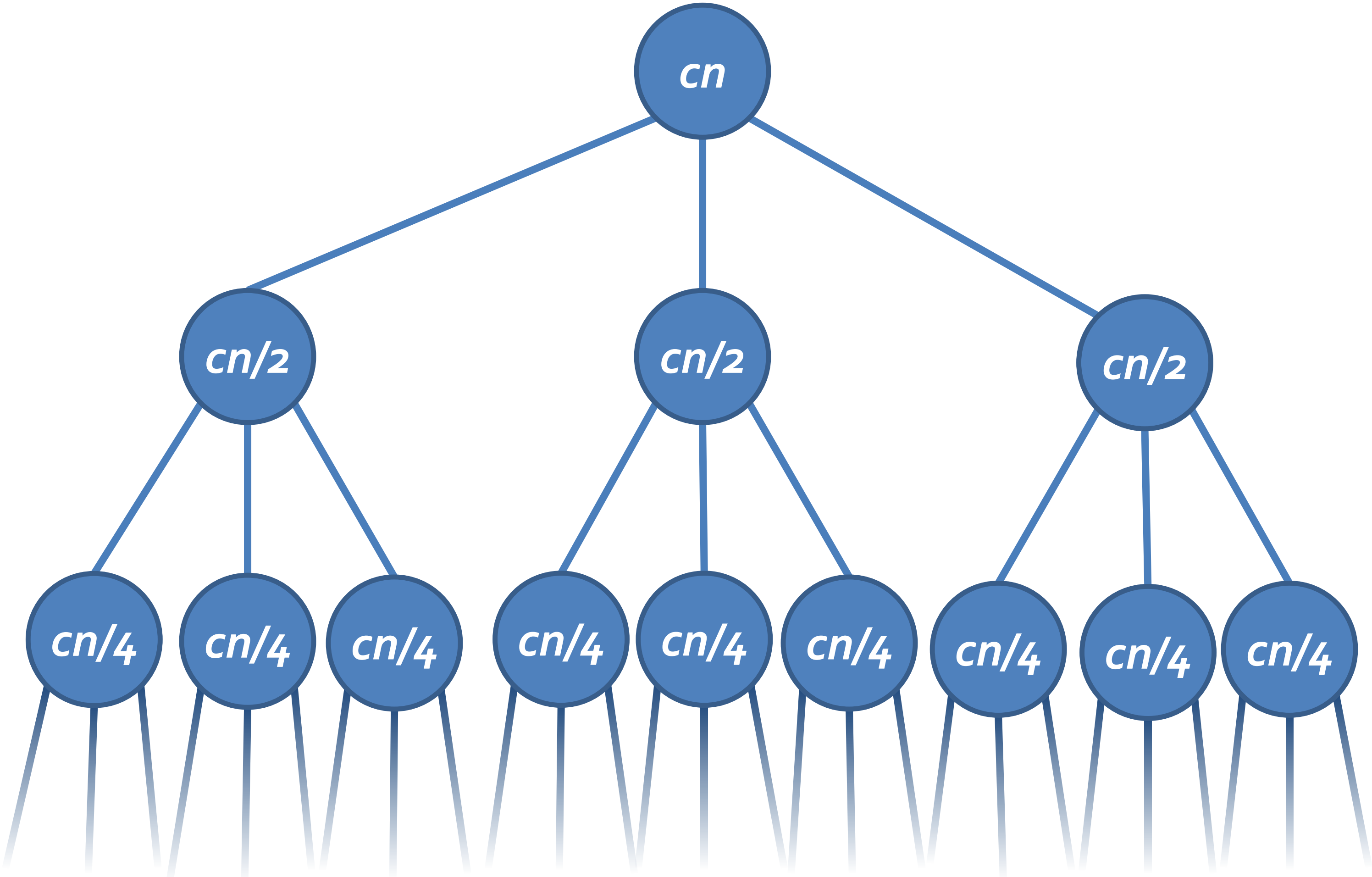
Further Recurrence Relations

Three-sentence Summary of Further Recurrence Relations

Further recurrence relations

- We have seen that recurrence relations of the form $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ are bounded by O(***n*** log ***n***)
- What about $T(n) \leq qT\left(\frac{n}{2}\right) + cn$ where ***q*** is bigger than 2 (more than two sub-problems)?
- There will still be $\log_2$***n*** levels of recursion
- However, there will not be a consistent ***cn*** amount of work at each level

Consider $q = 3$



cn

$(3/2)cn$

$(9/4)cn$

Converting to summation

- For $\boldsymbol{q}$ = 3, it's $T(n) \leq \sum_{j=0}^{\log_2 n-1} \left(\frac{3}{2}\right)^j cn$
- In general, it's

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j$$

- This is a geometric series, where $r = \frac{q}{2}$

$$T(n) \leq cn\left(\frac{r^{\log_2 n} - 1}{r-1}\right) \leq cn\left(\frac{r^{\log_2 n}}{r-1}\right)$$

Final bound

$$T(n) \le cn\left(\frac{r^{\log_2 n} - 1}{r - 1}\right) \le cn\left(\frac{r^{\log_2 n}}{r - 1}\right)$$

- Since $r - 1$ is a constant, we can pull it out
- $T(n) \le \left(\frac{c}{r-1}\right) n r^{\log_2 n}$
- For $a > 1$ and $b > 1$, $a^{\log b} = b^{\log a}$, thus $r^{\log_2 n} = n^{\log_2 r} = n^{\log_2 (q/2)} = n^{(\log_2 q) - 1}$
- $T(n) \le \left(\frac{c}{r-1}\right) n \cdot n^{(\log_2 q) - 1} \le \left(\frac{c}{r-1}\right) n^{\log_2 q}$ which is $O\left(n^{\log_2 q}\right)$

What about a single sub-problem?

- We will still have $\log_2 n - 1$ levels
- However, we'll cut our work in half each time

$$T(n) \leq T\left(\frac{n}{2}\right) + cn \leq \sum_{j=0}^{\log_2 n - 1} \left(\frac{1}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n - 1} \frac{1}{2^j}$$

- Summing all the way to infinity, $1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2$
- Thus, $T(n) \leq 2cn$ which is $O(n)$

What might that look like in code?

- Here's a non-recursive version in Java

```
int counter = 0;
for( int i = 1; i <= n; i *= 2 )
    for( int j = 1; j <= i; j++ )
        counter++;
```

- We've just shown that this is O(***n***), in spite of the two `for` loops

Quiz

Upcoming

Next time…

- Counting inversions

Reminders

- **Assignment 3 is due on Friday**
- Read section 5.3
- Extra credit opportunities (0.5% each):
  - Hristov teaching demo: 2/19 11:30-12:25 a.m. in Point 113
  - Hristov research talk: 2/19 4:30-5:30 p.m. in Point 139