Objectives

- Divide and conquer algorithms
  - Recurrence relations
  - Counting inversions

Review

- What approach are we learning to solve problems (as of Wednesday)?
- What is a recurrence relation?
- How can you compute D&C running times?

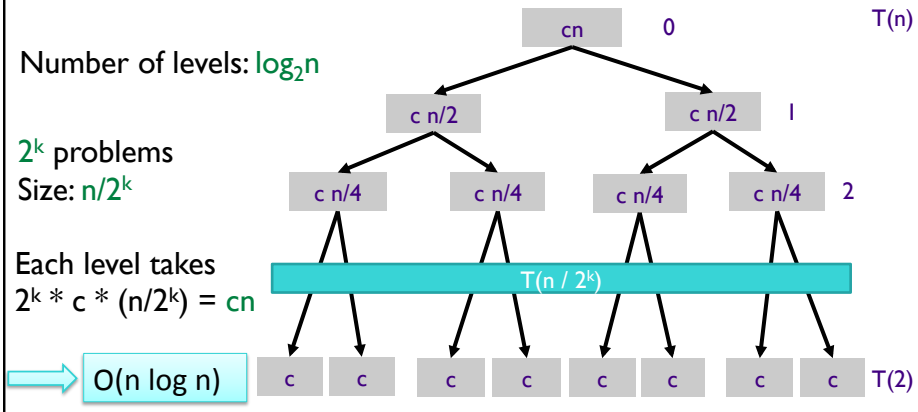
Review: Unrolling Recurrence

- How many levels are there (assuming *n* is a power of 2)?
- How much does each level cost, in terms of the level?
- What is the total run time?

Number of levels: $\log_2 n$

2^k problems
Size: $n/2^k$

Each level takes
$2^k * c * (n/2^k) = cn$



O(n log n)

Review: Approaches to Solving Recurrences

1. Unroll recursion
   - Look for patterns in runtime at each level
   - Sum up running times over all levels

2. Substitute guess solution into recurrence
   - Check that it works
   - Induction on n

Alternative: Proof by Induction

- Claim. If T(n) satisfies the recurrence 2 T(n/2) + *cn*, then T(n) = n $\log_2$ n.
- Pf. (by induction on n)
  - Base case: n = 2
  - Inductive hypothesis: T(n) ≤ cn $\log_2$ n
  - Goal: show that T(2n) = 2cn $\log_2$(2n)
    - Substitute *2n* in for *n* into the hypothesis

Why doubling n?

Proof by Induction

- Claim. If T(n) satisfies the recurrence T(n) = 2 T(n/2) + *cn*, then T(n) = n $\log_2$ n.
- Pf. (by induction on n)
  - Inductive hypothesis: T(n) ≤ cn $\log_2$ n
  - Goal: show that T(2n) = 2cn $\log_2$ (2n)

$$
\begin{aligned}
T(2n) &= 2T(n) + c2n \\
&= 2cn \log_2 n + 2cn \\
&= 2cn (\log_2(2n)-1) + 2cn \\
&= 2cn \log_2(2n) - 2cn + 2cn \\
&= 2cn \log_2(2n) \ \checkmark
\end{aligned}
$$

- Recurrence relation
- Replace T(n) w/ induction hypothesis
- Log rules: what is the difference between $\log_2(2n)$ and $\log_2(n)$?

Another Recurrence Relation: Binary Search

- How does binary search work?
- What is its recurrence relation?

Analyzing Binary Search

```
BinarySearch( L[1…n], key ):
     if len(L) == 1 and L[1] == key:
          return 1 # return the index
     else:
          return NOT_FOUND
     mid = n/2
     if L[mid] == key:
          return mid  # return the index
     if L[mid] < key:
          return BinarySearch(L[mid+1:], key)
     else:
          return BinarySearch(L[:mid], key)
```

What is the recurrence relation?

Analyzing Binary Search

```
BinarySearch( L[1…n], key ):
     if len(L) == 1 and L[1] == key:
          return 1 # return the index
     else:
          return NOT_FOUND
     mid = n/2
     if L[mid] == key:
          return mid  # return the index
     if L[mid] < key:
          return BinarySearch(L[mid+1:], key)
     else:
          return BinarySearch(L[:mid], key)
```

What is the recurrence relation?

$T(n) = T(n/2) + c$

Unroll the Recurrence

- $T(n) = T(n/2) + c$
- Which makes the runtime?

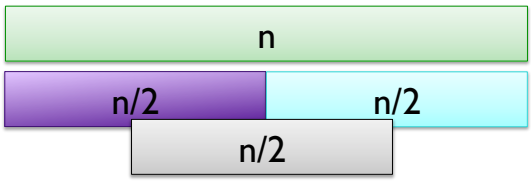
Unroll the Recurrence

- T(n) = T(n/2) + c
  - Constant work at each level
  - Number of levels: log n
- Which makes the runtime? O(log n)

Another Recurrence Relation

- Instead of recursively solving 2 problems, solve ***q*** problems
  - Size of problems is still n/2
- Combining solutions is still O(n)

Example: q=3:



What is the recurrence relation?

Another Recurrence Relation

- Instead of recursively solving 2 problems, solve ***q*** problems
  - Size of problems is still n/2
- Combining solutions is still O(n)
- Recurrence relation:
  - For some constant *c*,

    $T(n) \leq q\ T(n/2) + cn$ when $n > 2$

    $T(2) \leq c$

Intuition about running time?

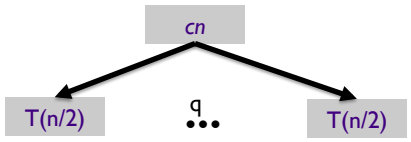
Unrolling Recurrence, q > 2

$T(n) \leq q\ T(n/2) + cn$

Unrolling Recurrence, q > 2

- First level:

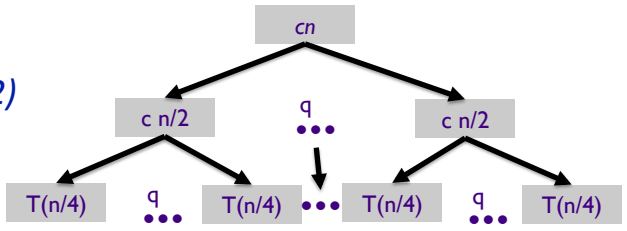
  q T(n/2) + *cn*



Unrolling Recurrence, q > 2

- Next level:

  q T(n/4) + *c(n/2)*

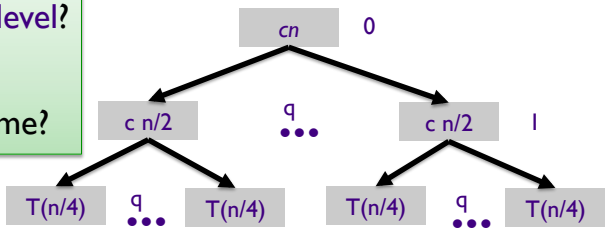


Unrolling Recurrence, q > 2

How much does each level cost, in terms of the level?

Number of levels?

What is the total run time?



q^k problems at level k
Size: $n/2^k$

Number of levels: $\log_2 n$

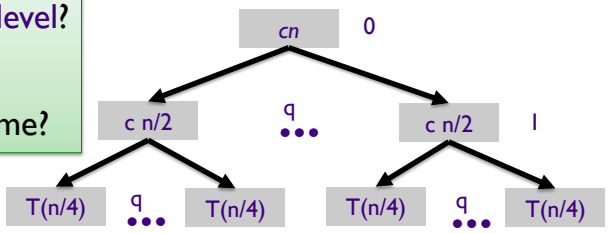
Each level takes $q^k * c * (n/2^k) = (q/2)^j$ cn
→ Total work per level is *increasing* as level increases

Unrolling Recurrence, q > 2

How much does each level cost, in terms of the level?

Number of levels?

What is the total run time?



$T(n) \leq \Sigma_{j=0,\log n} (q/2)^j$ cn

Geometric series:
(constant ratio between successive terms)
Multiplying previous term by (q/2)

→ $O(n^{\log_2 q})$

Unrolling the Recurrence

- Generalize: What are the steps?

Summary

- Use recurrences to analyze the runtime of divide and conquer algorithms
- Need to figure out
  - Number of sub problems
  - Size of sub problems
  - Number of times divided (number of levels)
  - Cost of merging problems

Know Your Recurrence Relations

What algorithm has this recurrence relation?
What is that algorithm's running time?

Recurrence	Algorithm	Running Time
T(n) = T(n/2) + O(1)		
T(n) = T(n-1) + O(1)		
T(n) = 2 T(n/2) + O(1)		
T(n) = T(n-1) + O(n)		
T(n) = 2 T(n/2) + O(n)		

Know Your Recurrence Relations

What algorithm has this recurrence relation?
What is that algorithm's running time?

Recurrence	Algorithm	Running Time
T(n) = T(n/2) + O(1)	Binary Search	O(log n)
T(n) = T(n-1) + O(1)	Sequential/ Linear Search	O(n)
T(n) = 2 T(n/2) + O(1)	Binary Tree Traversal	O(n)
T(n) = T(n-1) + O(n)	Selection Sort	$O(n^2)$
T(n) = 2 T(n/2) + O(n)	Merge Sort	O(n log n)

COUNTING INVERSIONS

Comparing Rankings

- To determine similarity of rankings, need a metric
- Similarity metric: number of inversions between two rankings
  - My rank: 1, 2, …, n
  - Your rank: $a_1, a_2, \ldots, a_n$
  - Movies i and j *inverted* if i < j but $a_i > a_j$

Movies

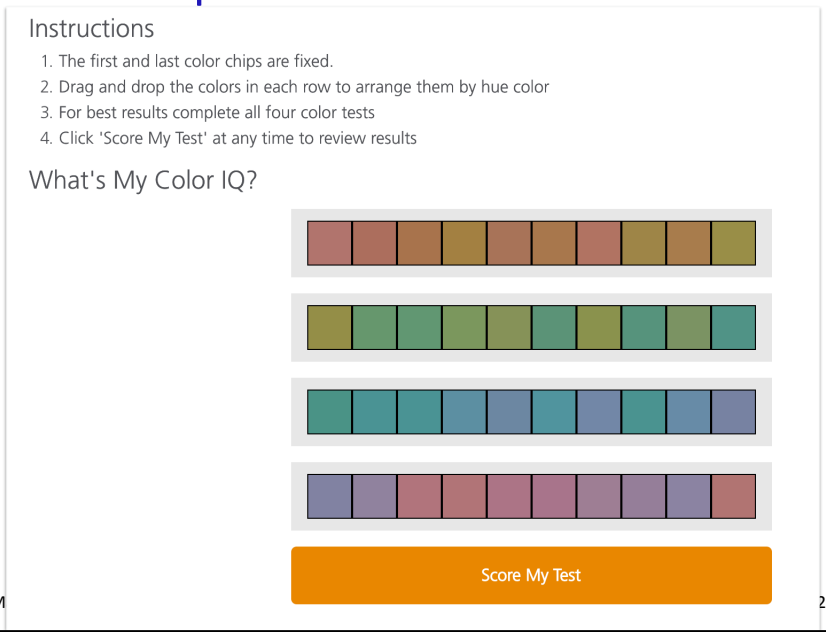
	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions:

3-2, 4-2

https://www.xrite.com/hue-test

Color Comparison Test



M 25

Comparing Rankings

- To determine similarity of rankings, need a metric
- Similarity metric: number of inversions between two rankings
  - My rank: 1, 2, …, n
  - Your rank: $a_1, a_2, \ldots, a_n$
  - Movies i and j *inverted* if i < j but $a_i > a_j$

Naïve/Brute force solution?

Movies

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions:

3-2, 4-2

Brute Force Solution

- Look at every pair (i,j) and determine if they are an inversion
- Requires $\Theta(n^2)$ time
  - Note: Already an efficient algorithm but try to improve upon runtime

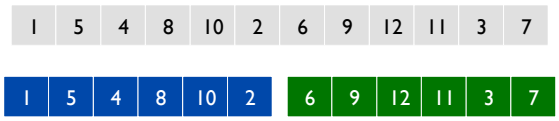
Towards a Better Solution…
- Can't look at each inversion individually

Counting Inversions: Divide-and-Conquer

Assume number represents where item *should* be in the list, i.e., where it is in someone else's list

Counting Inversions: Divide-and-Conquer

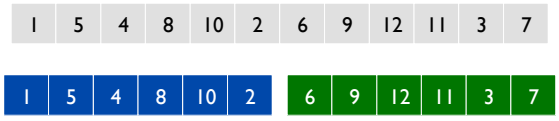
- Divide: separate list into two pieces



Divide: O(1)

Counting Inversions: Divide-and-Conquer

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half



Divide: O(1)

What is the recurrence relation so far?

5 blue-blue inversions

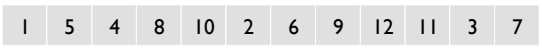
5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half



Divide: O(1)

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

5-4, 5-2, 4-2, 8-2, 10-2

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

What do we need to do next?

Counting Inversions: Divide-and-Conquer

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities

Divide: O(1)

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

Combine cost: ???

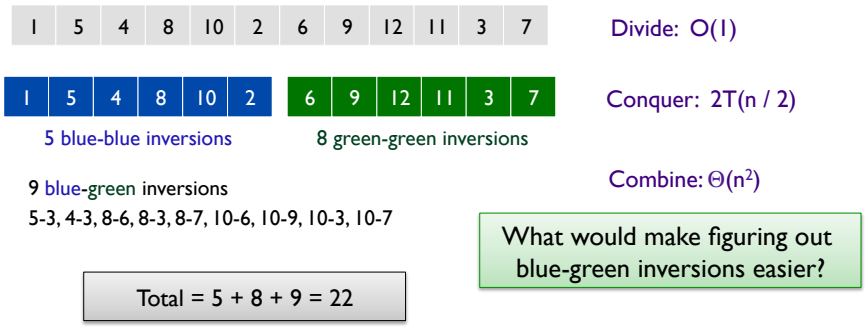
9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Total = 5 + 8 + 9 = 22

Counting Inversions: Divide-and-Conquer

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities



Looking Ahead

- Wiki: 4.8, 5.1-5.3
- Problem Set 7
  - One of the harder problem sets