Objectives

- Review Huffman Codes
- Introducing Divide and Conquer Algorithms

Huffman Codes

- What problem is Huffman codes trying to solve?
- What is the solution?
  - What metric are we using for optimality?
  - Algorithm, outcome/data structures generated
  - What is the intuition behind the solutions?
- Algorithm analysis
  - How is it greedy?
  - What data structures do we need to use?

Reviewed from previous day's slides

DIVIDE AND CONQUER ALGORITHMS

Divide-and-Conquer

Divide et impera.
Veni, vidi, vici.
- *Julius Caesar*

- Divide-and-conquer process
  - **Break up** problem into **several parts**
  - Solve each part **recursively**
  - **Combine** solutions to sub-problems into overall solution
- Most common usage:
  - Break up problem of size n into two equal parts of size ½n
  - Solve two parts recursively
  - Combine two solutions into overall solution

Discussion

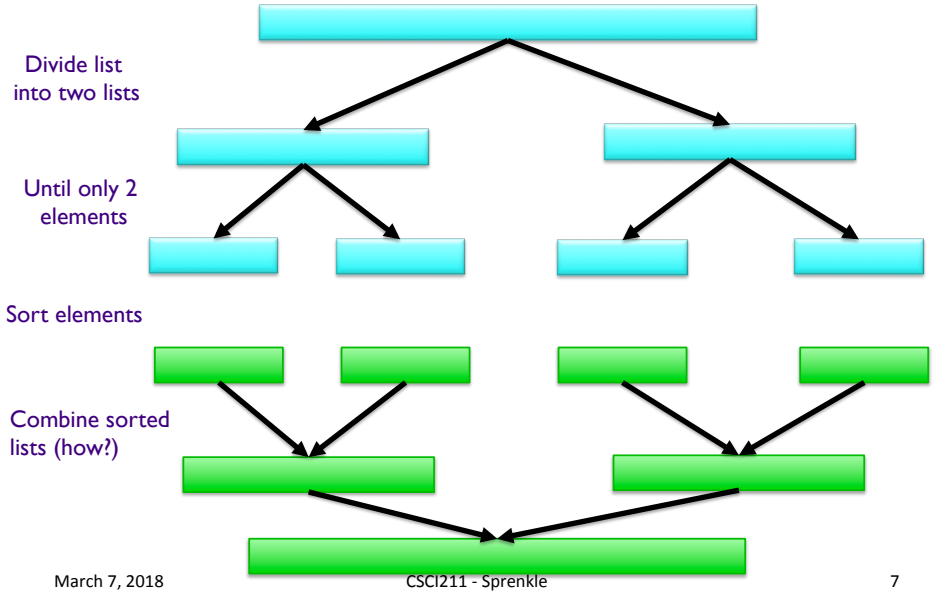
- What is a well-known divide and conquer algorithm?

Merge Sort

Merge Sort

- How does Merge Sort work?
- When do we stop?

Merge Sort



RECURRENCE RELATIONS

Analyzing Merge Sort

General Template
- Break up problem of size n into two equal parts of size ½n
- Solve two parts recursively
- Combine two solutions into overall solution

- **Def.** T(n) = number of comparisons to mergesort an input of size *n*
- Want to say a bit more about what T(n) is
  - Break it down more…

What can we say about the running time w.r.t. to the different parts of the above template?

Analyzing Merge Sort

General Template
- Break up problem of size n into two equal parts of size ½n **O(1)**
- Solve two parts recursively **T(n/2) + T(n/2)**
- Combine two solutions into overall solution **O(n)**

- **Def.** T(n) = number of comparisons to mergesort an input of size *n*
- Want to say a bit more about what T(n) is
  - Break it down more…

What is the base case? Its running time?

Merge Sort's Recurrence Relation

```
MergeSort( L[1…n] ):
     if len(L) == 1:
          return L
     if len(L) == 2:
          compare the two entries in L,
          swap if necessary
          return L
     A = MergeSort(L[:n/2])
     B = MergeSort(L[n/2+1:])
     M = Merge(A, B)
     return M
```

Base cases

T(n/2)

T(n/2)

O(n)

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

Merge Sort's Recurrence Relation

- Put an *upperbound* on T(n):

O(n)

For some constant c,

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

$T(2) \leq c$

Solve T(n) to come up with explicit bound

Approaches to Solving Recurrences

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

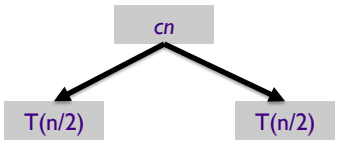
2. Substitute guess solution into recurrence
   - Check that it works
   - Induction on n

Unrolling Recurrence: T(n)

$T(n) = 2\,T(n/2) + cn$

Unrolling Recurrence: 2 T(n/2) + cn

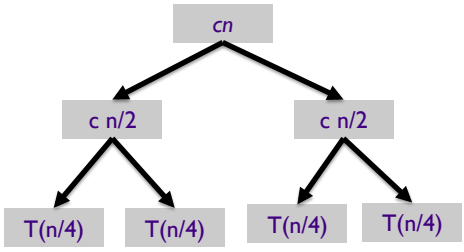
- First level: 2 T(n/2) + *cn*



How does the next level break down?

Unrolling Recurrence: 2 T(n/2) + cn

- Next level:



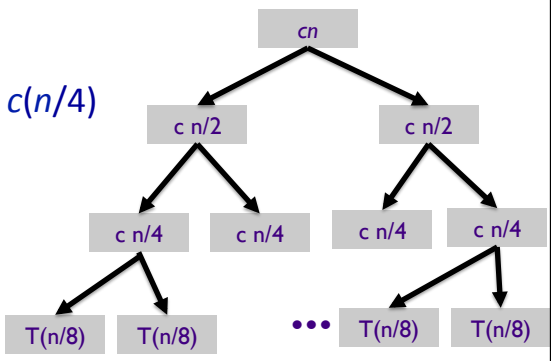
Each one is 2 T(n/4) + *c*(*n*/2)

Next level?

Unrolling Recurrence

- Next level:

  Each one is 2 T(n/8) + $c(n/4)$

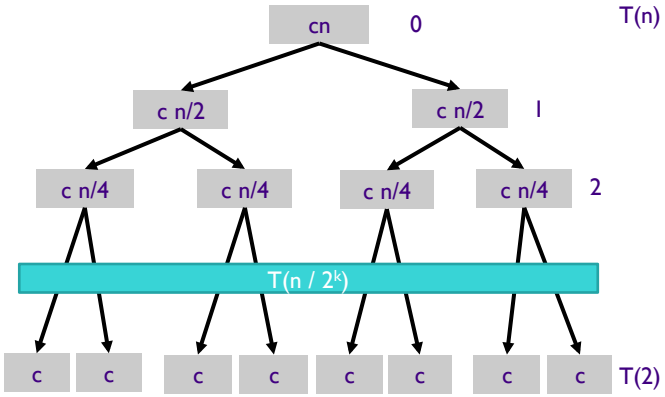


And so on…

What does the final level look like?

Unrolling Recurrence

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



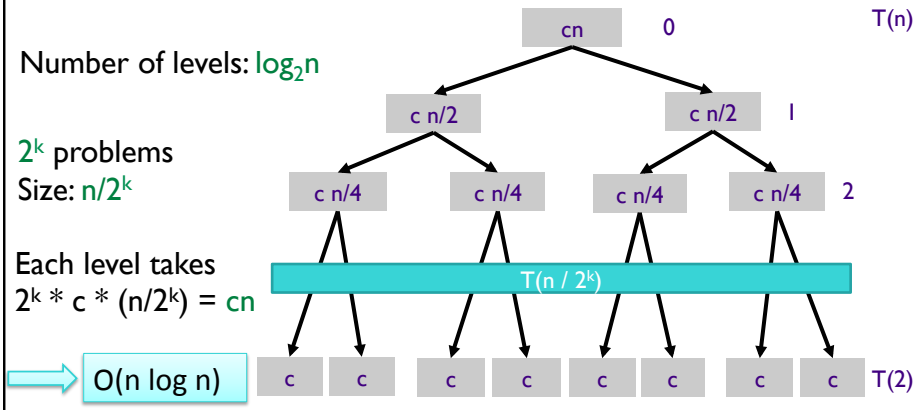
Unrolling Recurrence

- 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)

Looking Ahead

- Problem Set 6 due Friday