Objectives

Registrar Review

Algorithm Approach: Divide and Conquer

- Recurrence Relations
- Algorithm development

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

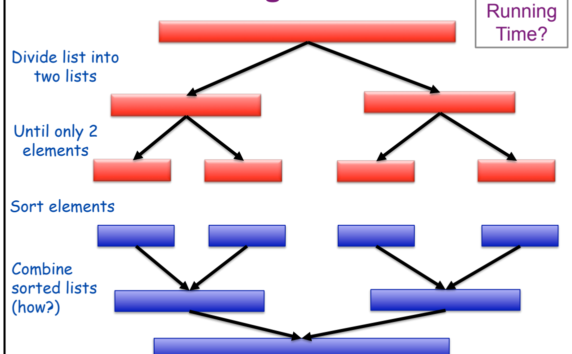
Merge Sort

How does Merge Sort work?

When do we stop?

Merge Sort

Costs?
Running
Time?



RECURRENCE RELATIONS

Analyzing Merge Sort

General Template
- Break up problem of size n into two equal parts of size $\frac{1}{2}$n
- 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 $\frac{1}{2}$n
- 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 can we say about the running time w.r.t. to the different parts of the above template?

Merge Sort's Recurrence Relation

Put an *upperbound* on T(n):

For some constant *c*,

$T(n) \le 2\,T(n/2) + \underbrace{cn}_{O(n)}$ when n > 2,

$T(2) \le c$.

Solve T(n) to come up with explicit bound

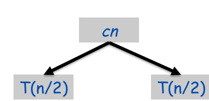
Approaches to Solving Recurrences

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

Unrolling Recurrence

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

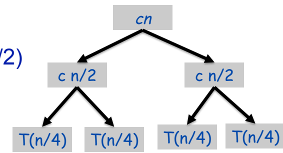


How does the next level break down?

Unrolling Recurrence

Next level:

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

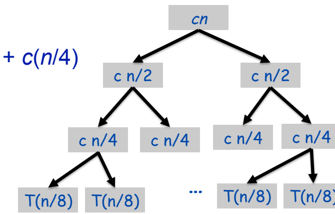


Next level?

Unrolling Recurrence

Next level:

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



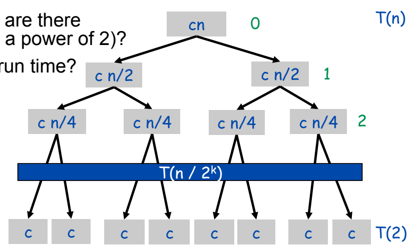
And so on…

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

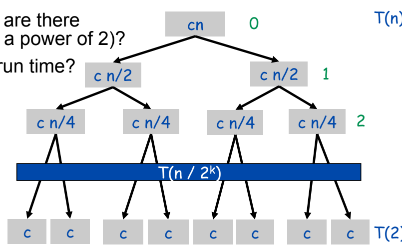
How much does each level cost, in terms of the level?

How many levels are there (assuming *n* is a power of 2)?

What is the total run time?

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

Number of levels: $\log_2 n$



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

Alternative: Proof by Induction

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1
- Inductive hypothesis: T(n) = n $\log_2$ n
- Goal: show that T(2n) = 2n $\log_2$ (2n)

Why doubling n?

Proof by Induction

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Inductive hypothesis: T(n) = n $\log_2$ n

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

Another Example

Instead of recursively solving 2 problems, solve *q* problems

- Size of problems is still n/2

Combining solutions is still O(n)

Another Example

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 $n > 2$

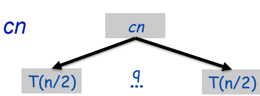
  $T(2) \leq c$

Intuition about running time?

Unrolling Recurrence, q > 2

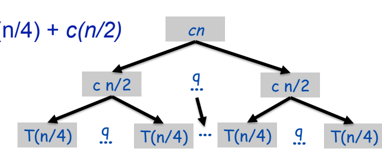
Unrolling Recurrence, q > 2

First level: $q\,T(n/2) + cn$



Unrolling Recurrence, q > 2

Next level: $q\,T(n/4) + c(n/2)$

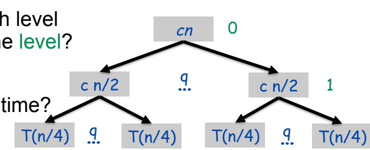


Unrolling Recurrence, q > 2

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: $\log_2 n$

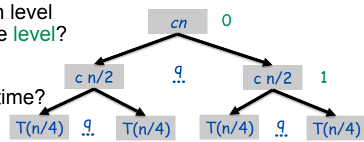
Each level takes $q^k * c * (n/2^k) = (q/2)^j\,cn$
→ Total work per level is *increasing* as level increases

Unrolling Recurrence, q > 2

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

Number of levels?

What is the total run time?



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

Geometric series: → $O(n^{\log_2 q})$

Summary

Use recurrences to analyze the run time of divide and conquer algorithms

- Number of sub problems
- Size of sub problems
- Number of times divided (number of levels)
- Cost of merging problems

COUNTING INVERSIONS

Problem Context

Movie site tries to match your song preferences with others

- You rank *n* movies
- Movie site consults database to find people with similar tastes
  - **Collaborative filtering**

Meta-search tools

- Do same query on several search engines
- Synthesize results by looking for similarities and differences in search engines' results rankings

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

What are the 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 $a_i > a_j$

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

Applications

Voting theory

Collaborative filtering

Measuring the "sortedness" of an array

Sensitivity analysis of Google's ranking function

Rank aggregation for meta-searching on the Web

Nonparametric statistics (e.g., Kendall's Tau distance)

Forming a Better Solution

Better than brute force $\Theta(n^2)$

- Can't look at each inversion individually

Continued on Friday …