Advanced Counting

Dr. Isaac Griffith

Idaho State University

Outline

The lecture is structured as follows:

- Applications of Recurrence Relations
- Divide-and-Conquer Algorithms and Recurrence Relations

ROAR

Recurrence Relations

- **Definition:** A *recurrence relation* for a sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more previous elements $a_0, \ldots a_{n-1}$ of the sequence, for all $n \geq n_0$

Recurrence Relations

- **Definition:** A *recurrence relation* for a sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more previous elements $a_0, \ldots a_{n-1}$ of the sequence, for all $n \geq n_0$
- **Definition:** A particular sequence (described non-recursively) is said to solve the given recurrence relation if it is consistent with the definition of the recurrence.
  - A given recurrence relation may have many solutions

Recurrence Relations

- **Definition:** A *recurrence relation* for a sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more previous elements $a_0, \ldots a_{n-1}$ of the sequence, for all $n \geq n_0$

- **Definition:** A particular sequence (described non-recursively) is said to solve the given recurrence relation if it is consistent with the definition of the recurrence.
  - A given recurrence relation may have many solutions

- Such relations can be used for many things:
  - Study and solve counting problems beyond the basic counting techniques discussed previously
  - Modeling a wide variety of problems, including
    - Compound Interest
    - Counting rabbits on an island
    - Tower of Hanoi
  - Additionally, when algorithms such as Mergesort, we can use *divide-and-conquer recurrence relations* to evaluate the time complexity of this class of algorithms

Rabbits and Fibonacci Numbers

- A young pair of rabbits (one of each sex) is placed on an island.
  - A pair of rabbits does not breed until they are 2 months old.
  - After they are 2 months old, each pair of rabbits produces another pair each month

Month	Reproducing Pairs	Young Pairs	Total Pairs
1	0	1	1
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8

Rabbits and Fibonacci Numbers

- A young pair of rabbits (one of each sex) is placed on an island.
  - A pair of rabbits does not breed until they are 2 months old.
  - After they are 2 months old, each pair of rabbits produces another pair each month

Month	Reproducing Pairs	Young Pairs	Total Pairs
1	0	1	1
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8

- Find a recurrence for the number of pairs of rabbits on the island after n months, assuming that no rabbits ever die.

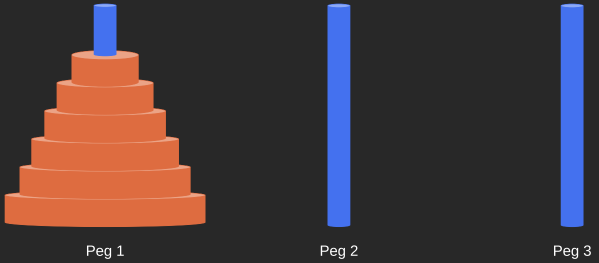
Rabbits and Fibonacci Numbers

- **Solution:** Denote by f_n the number of pairs of rabbits after n months.
  - We can see that f_n, for $n = 1, 2, 3, \ldots$, are the terms of the Fibonacci sequence
  - After the first month the number of pairs is $f_1 = 1$
  - After the second month the number of pairs is $f_2 = 1$
  - To find the number of pairs after n months simply add the number on the island from the previous month, f_{n-1}, and the number of newborn pairs, $f_{n - 2}
  - The sequence $\{f_n\}$ satisfies the following recurrence relation:

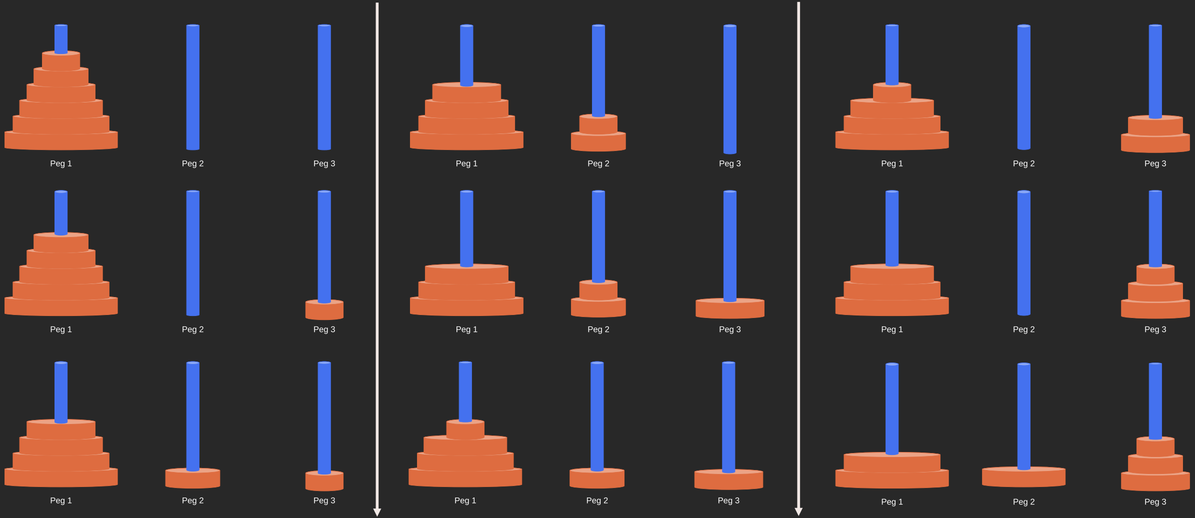
$$f_n = f_{n-1} + f_{n-2}, \;\; f_1 = 1 \; f_2 = 1$$

Tower of Hanoi

- **Problem:** Get all disks from peg 1 to peg 2
  - Only move 1 disk at a time
  - Never set a larger disk on top of a smaller one



Tower of Hanoi



Tower of Hanoi

- **Solution:** Let H_n = number of moves for a stack of n disks

  Optimal strategy:

  - Move top $n-1$ disks to spare peg. (H_{n-1} moves)
  - Move bottom disk. (1 move)
  - Move top $n-1$ disks to bottom disk. (H_{n-1} moves)

  Note: $H_n = 2H_{n-1} + 1$

Tower of Hanoi

Solution: Using the Iterative Method

$$
\begin{aligned}
H_n &= 2h_{n-1} + 1 \\
&= 2(2H_{n-2} + 1) + 1 = 2^2 H_{n-2} + 2 + 1 \\
&= 2^2(2H_{n-3} + 1) + 2 + 1 = 2^3 H_{n-3} + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^{n-1} H_1 + 2^{n-2} + \ldots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 \text{ (since } H_1 = 1) \\
&= \sum_{i=0}^{n-1} 2^i \\
&= 2^n - 1
\end{aligned}
$$

Catalan Numbers

- **Example:** Find a recurrence relation for C_n, the number of ways to parenthesize the product of $n+1$ numbers, $x_0 \cdot x_1 \cdot x_2 \cdot \cdots \cdot x_n$, to specify the order of multiplication.
  - For example: $C_3 = 5$ because there are 5 ways to parenthesize $x_0 \cdot x_1 \cdot x_2 \cdot x_3$ to determine the order of multiplication:

$$((x_0 \cdot x_1)) \cdot x_2) \cdot x_3) \qquad (x_0 \cdot (x_1 \cdot x_2) \cdot x_3) \qquad (x_0 \cdot x_1) \cdot (x_2 \cdot x_3)$$
$$x_0 \cdot ((x_1 \cdot x_2) \cdot x_3) \qquad x_0 \cdot (x_1 \cdot (x_2 \cdot x_3))$$

Catalan Numbers

- **Solution:**
  - We note that however parentheses are inserted, one "·" operator always remains outside all parentheses, the final operator
  - This operator occurs between two of the n_1 numbers, say x_k and x_{k+1}
  - There are then $C_k C_{n-k-1}$ ways to insert parentheses
  - Because this final operator can fall between any two of the numbers it follows that

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \ldots + C_{n-2} C_1 + C_{n-1} C_0 = \sum_{k=0}^{n-1} C_k C_{n-k-1}$$

Catalan Numbers

- **Solution:**
  - We note that however parentheses are inserted, one "$\cdot$" operator always remains outside all parentheses, the final operator
  - This operator occurs between two of the n_1 numbers, say x_k and x_{k+1}
  - There are then $C_k C_{n-k-1}$ ways to insert parentheses
  - Because this final operator can fall between any two of the numbers it follows that

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \ldots + C_{n-2} C_1 + C_{n-1} C_0 = \sum_{k=0}^{n-1} C_k C_{n-k-1}$$

- The sequence $\{C_n\}$ is the sequence of **Catalan Numbers**, where $C_n = C(2n, n)/(n+1) \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$

Dynamic Programming

- **Dynamic Programming:** an algorithmic paradigm used in solving optimization problems efficiently. It uses recursion to subdivide a problem into smaller or simpler overlapping subproblems

Dynamic Programming

- **Dynamic Programming:** an algorithmic paradigm used in solving optimization problems efficiently. It uses recursion to subdivide a problem into smaller or simpler overlapping subproblems

- Problems to which Dynamic Programming is applicable must have the following characteristics:
  - Problem is easily subdivided into simpler subproblems whose aggregate is a solution to the larger problem.
  - The subproblems overlap, that is the same subproblem will be seen more than once, and we can take advantage of storing the results of calculations for later use in similar subproblems.
  - The process of storing and reusing the results of a calculation is called **Memoization** (it is akin to looking up values in a table)

Dynamic Programming

- **Dynamic Programming:** an algorithmic paradigm used in solving optimization problems efficiently. It uses recursion to subdivide a problem into smaller or simpler overlapping subproblems

- Problems to which Dynamic Programming is applicable must have the following characteristics:
  - Problem is easily subdivided into simpler subproblems whose aggregate is a solution to the larger problem.
  - The subproblems overlap, that is the same subproblem will be seen more than once, and we can take advantage of storing the results of calculations for later use in similar subproblems.
  - The process of storing and reusing the results of a calculation is called **Memoization** (it is akin to looking up values in a table)

- This approach was introduced by Richard Bellman in the 1950s.

Dynamic Programming

- **Example:** Suppose we have n talks
  - where talk j has the following properties
    - Begins at time t_j
    - Ends at time e_j
    - Attended by w_j students
  - We want a schedule that maximizes the sum of w_j
  - We denote $T(j)$ as the maximum attendees for an optimal schedule from the first j talks
    - $T(n)$ is then the maximal number of total attendees for an optimal schedule of n talks

Dynamic Programming

- Initial idea:
  1. Sort the talks in order of increasing end time, and renumber them such that:

$$e_1 \leq e_2 \leq \ldots \leq e_n$$

    - Two talks are deemed *compatible* if they can be on the same schedule without overlapping.

Dynamic Programming

- Initial idea:
  1. Sort the talks in order of increasing end time, and renumber them such that:

$$e_1 \leq e_2 \leq \ldots \leq e_n$$

     - Two talks are deemed *compatible* if they can be on the same schedule without overlapping.

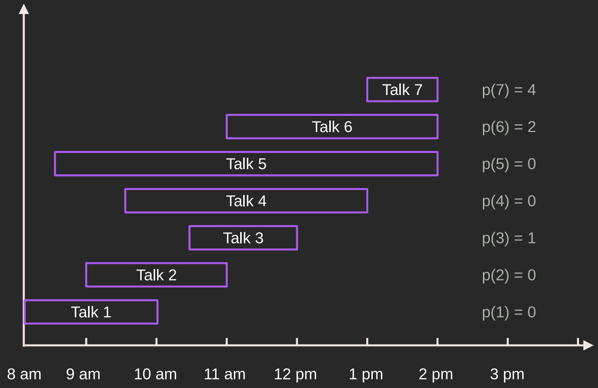
  2. Define $p(j)$

$$p(j) = \begin{cases} \text{largest } i & i < j \text{ for which } e_i \leq s_j \\ 0 & \text{otherwise (no such talk exists)} \end{cases}$$

Dynamic Programming

- Consider the following talks:
  1. 8 am – 10 am
  2. 9 am – 11 am
  3. 10:30 am – 12 pm
  4. 9:30 am – 1:00 pm
  5. 8:30 am – 2:00 pm
  6. 11:00 am – 2:00 pm
  7. 1:00 pm – 2:00 pm

  - Find $p(j)$ for $j = 1, 2, \ldots, 7$



Dynamic Programming

- To construct a dynamic programming algorithm, we first need a recurrence relation
  - Note if $j \leq n$, there are two possibilities for an optimal schedule of the first j talks
    (i) talk j belongs to the optimal schedule
    (ii) it does not

Dynamic Programming

- To construct a dynamic programming algorithm, we first need a recurrence relation
  - Note if $j \leq n$, there are two possibilities for an optimal schedule of the first j talks
    (i) talk j belongs to the optimal schedule
    (ii) it does not

- *Case (i):* talks $p(j) + 1, \ldots, j - 1$ do not belong to the schedule. The other talks in the schedule must comprise an optimal schedule of talks $1, 2, \ldots, p(j)$.
  - Thus, we have $T(j) = w_j + T(p(j))$

Dynamic Programming

- To construct a dynamic programming algorithm, we first need a recurrence relation
  - Note if $j \leq n$, there are two possibilities for an optimal schedule of the first j talks
    (i) talk j belongs to the optimal schedule
    (ii) it does not

- *Case (i):* talks $p(j) + 1, \ldots, j - 1$ do not belong to the schedule. The other talks in the schedule must comprise an optimal schedule of talks $1, 2, \ldots, p(j)$.
  - Thus, we have $T(j) = w_j + T(p(j))$

- *Case (ii):* When talk j does not belong to the optimal schedule, it follows that an optimal schedule from talks $1, 2, \ldots, j$ is the same as one from talks $1, 2, \ldots, j - 1$.
  - Thus, we have $T(j) = T(j - 1)$

Dynamic Programming

Idaho State University | Computer Science

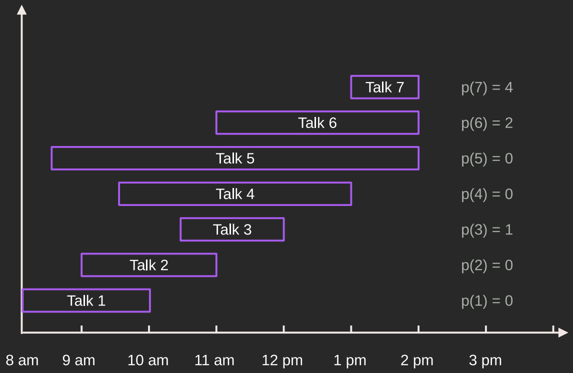
- We combine the cases and get the following recurrence relation:

$$T(j) = \max(w_j + T(p(j)), T(j-1))$$

- **Example:** Assume $w_1 = w_2 = w_3 = 25, w_4 = w_5 = 50$, and $w_6 = w_7 = 75$

$$\begin{aligned}
T(1) &= \max(25 + T(0), T(0)) = \max(25, 0) \\
&= 25 \text{ // store for later} \\
T(2) &= \max(25 + T(0), T(1)) = \max(25, 25) \\
&= 25 \text{ // retrieve T(1), store T(2)} \\
T(3) &= \max(25 + T(1), T(2)) = \max(50, 25) \\
&= 50 \text{ // retrieve T(1),T(2), store T(3)} \\
T(4) &= \max(50 + T(0), T(3)) = \max(50, 50) \\
&= 50 \text{ // retrieve T(3), store T(4)} \\
T(5) &= \max(50 + T(0), T(4)) = \max(50, 50) \\
&= 50 \text{ // retrieve T(4), store T(5)} \\
T(6) &= \max(75 + T(2), T(5)) = \max(75 + 25, 50) \\
&= 100 \text{ // retrieve T(2),T(5), store T(6)} \\
T(7) &= \max(75 + T(4), T(6)) = \max(75 + 50, 100) \\
&= 125 \text{ // retrieve T(4),T(6), store T(7)}
\end{aligned}$$

Solution: : 125 students will attend for the optimal schedule (1 -> 3 -> 7 or 4 -> 7)



ROAR

Dynamic Programming

- We combine the cases and get the following recurrence relation:

$$T(j) = \max(w_j + T(p(j)), T(j-1))$$

Dynamic Programming

- We combine the cases and get the following recurrence relation:

$$T(j) = \max(w_j + T(p(j)), T(j-1))$$

- To make this efficient we store the value of $T(j)$ after we compute it.
  - In most dynamic programming this type of calculation tends to be highly complex and thus costly to execute
  - So we store it and when the same subproblem occurs again, rather than recompute it we simply look up the value. This process is called **memoization**

Dynamic Programming

Idaho State University | Computer Science

- We combine the cases and get the following recurrence relation:

$$T(j) = \max(w_j + T(p(j)), T(j-1))$$

- To make this efficient we store the value of $T(j)$ after we compute it.
  - In most dynamic programming this type of calculation tends to be highly complex and thus costly to execute
  - So we store it and when the same subproblem occurs again, rather than recompute it we simply look up the value. This process is called **memoization**

- In this case, without *memoization* the algorithm would have an exponential worst-case complexity.

Dynamic Programming

The Algorithm for Maximum Attendees of a Optimal Schedule

procedure MAXATTENDEES($s_1, s_2, \ldots, s_n$: start times, $e_1, e_2, \ldots, e_n$: end times, $w_1, w_2, \ldots, w_n$: number of attendees)

    ▷ sort talks by end times and relabel so that $e_1 \leq e_2 \leq \ldots \leq e_n$
  for $j := 1$ **to** n **do**
    if no job i with $i < j$ is compatible with job i **then**
      $p(j) := 0$
    else $p(j) :=$ max   ▷ $i - i < j$ and job i is compatible with job j
    $T(0) := 0$
  for $j := 1$ **to** n **do**
    $T(j) :=$ max$(w_j + T(p(j)), T(j-1))$
  return $T(n)$

ROAR

Divide-and-Conquer Algorithms

CS 1187

Introduction

- Many types of problems are solvable by reducing a problem of size n into some number of a independent subproblems, each of size $\leq \lceil \frac{n}{b} \rceil$, where $a \geq 1$ and $b > 1$

Introduction

- Many types of problems are solvable by reducing a problem of size n into some number of a independent subproblems, each of size $\leq \lceil \frac{n}{b} \rceil$, where $a \geq 1$ and $b > 1$
- The time complexity to solve such problems is given by a recurrence relation

$$f(n) = af\left(\left\lceil \frac{n}{b} \right\rceil\right) + g(n)$$

Introduction

- Many types of problems are solvable by reducing a problem of size n into some number of a independent subproblems, each of size $\leq \lceil \frac{n}{b} \rceil$, where $a \geq 1$ and $b > 1$
- The time complexity to solve such problems is given by a recurrence relation

$$f(n) = af\left(\left\lceil \frac{n}{b} \right\rceil\right) + g(n)$$

- Such a recurrence is called a **divide-and-conquer recurrence relation**

Basic Analysis

- **Theorem 1:** Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + c$$

whenever n is divisible by b, where $a \geq 1$, b, is an integer greater than 1, and c is a positive real number. Then

$$f(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > 1 \\ O(\log n) & \text{if } a = 1 \end{cases}$$

Furthermore, when $n = b^k$ and $a \neq 1$, where k is a positive integer,

$$f(n) = C_1 n^{\log_b a} + C_2$$

where $C_1 = f(1) + c/(a-1)$ and $C_2 = -c/(a-1)$

Basic Analysis Example

Example: Let $f(n) = 5f(n/2) + 3$ and $f(1) = 7$. Find $f(2^k)$, where k is a positive integer. Also, estimate $f(n)$ if f is an increasing function.

ROAR

Basic Analysis Example

Example: Let $f(n) = 5f(n/2) + 3$ and $f(1) = 7$. Find $f(2^k)$, where k is a positive integer. Also, estimate $f(n)$ if f is an increasing function.

Solution: With $a = 5$, $b = 2$, and $c = 3$, we see that if $n = 2^k$, then

$$\begin{aligned} f(n) &= a^k\left[f(1) + c/(a-1)\right] + \left[-c/(a-1)\right] \\ &= 5^k\left[7 + (3/4)\right] - 3/4 \\ &= 5^k\,(31/4) - 3/4 \end{aligned}$$

By Theorem 1, we know that $f(n)$ is $O(n^{\log_b a}) = O(n^{\log 5})$

ROAR

Exercises

Exercise: Give the big-O estimate for $f(n) = f(n/2) + 1$ if f is an increasing function

Exercises

Exercise: Give the big-O estimate for $f(n) = f(n/2) + 1$ if f is an increasing function

- **Solution:** By Theorem 1, $f(n)$ is $O(\log n)$

Exercises

Exercise: Give the big-O estimate for $f(n) = f(n/2) + 1$ if f is an increasing function

- **Solution:** By Theorem 1, $f(n)$ is $O(\log n)$

Exercise: Give the big-O estimate for $f(n) = 2f(n/3) + 4 if *f* is an increasing function

Exercises

Idaho State University | Computer Science

Exercise: Give the big-O estimate for $f(n) = f(n/2) + 1$ if f is an increasing function

- **Solution:** By Theorem 1, $f(n)$ is $O(\log n)$

Exercise: Give the big-O estimate for $f(n) = 2f(n/3) + 4 if *f* is an increasing function

- **Solution:** By Theorem 1, $f(n)$ is $O(n^{\log_3 2})$

ROAR

The Master Theorem

- **The Master Theorem:** Consider a function $f(n)$ that, for all $n = b^k$, for all $n = b^k$ for all $k \in \mathbb{Z}^+$, satisfies the recurrence relating

$$f(n) = af\left(\frac{n}{b}\right) + cn^d$$

with $a \geq 1$, integer $b > 1$, real $c > 0$, $d \geq 0$. Then.

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Binary Search

Algorithm:

procedure BINSEARCH(A, x)
  $i := 1$
  $j := n$
  while $i < j$ **do**
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > A_m$ **then** $i := m + 1$
    else $j := m$
  if $x = A_j$ **then** $location := i$
  else $location := 0$
  return $location$

9		1	2	3	5	6	7	8	9
9		6	7	8	9				
9		8	9						
9		9							

Binary Search

Idaho State University | Computer Science

Algorithm:

procedure BINSEARCH(A, x)
- $i := 1$
- $j := n$
- **while** $i < j$ **do**
  - $m := \lfloor (i+j)/2 \rfloor$
  - **if** $x > A_m$ **then** $i := m + 1$
  - **else** $j := m$
- **if** $x = A_j$ **then** $location := i$
- **else** $location := 0$
- **return** $location$

9	1 2 3 5 6 7 8 9
9	6 7 8 9
9	8 9
9	9

Recurrence Relation:

- Break list into 1 sub-problem (smaller list) (so $a = 1$) of size $\leq \lceil \frac{n}{2} \rceil$ (so $b = 2$) or simply $n/2$ when n is even.
- Additionally, two comparison are needed to implement this reduction ($g(n) = 2$)
  - One to determine which half of the list and one to determine if any terms of the list remain

$$f(n) = f\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2$$

ROAR

Binary Search

Algorithm:

procedure BINSEARCH(A, x)
- $i := 1$
- $j := n$
- **while** $i < j$ **do**
  - $m := \lfloor (i + j)/2 \rfloor$
  - **if** $x > A_m$ **then** $i := m + 1$
  - **else** $j := m$
- **if** $x = A_j$ **then** $location := i$
- **else** $location := 0$
- **return** $location$

9		1	2	3	5	6	7	8	9
9		6	7	8	9				
9		8	9						
9		9							

Recurrence Relation:

- Break list into 1 sub-problem (smaller list) (so $a = 1$) of size $\leq \lceil \frac{n}{2} \rceil$ (so $b = 2$) or simply $n/2$ when n is even.
- Additionally, two comparison are needed to implement this reduction ($g(n) = 2$)
  - One to determine which half of the list and one to determine if any terms of the list remain

$$f(n) = f\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2$$

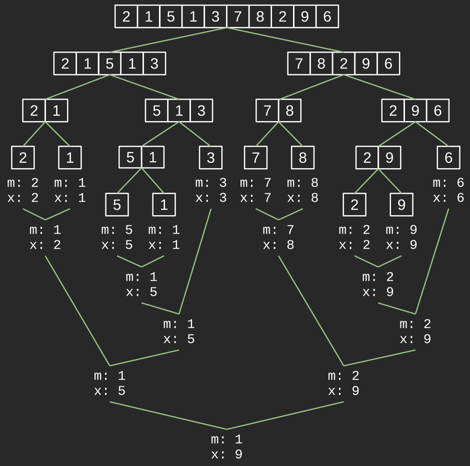
Complexity (from Theorem 1):

- Since the recurrence relation is $f(n) = f(n/2) + 2$ when n is even
- f is then the number of comparisons required to perform a binary search on a sequence of size n
- By Theorem 1, the time complexity of $f(n)$ is $O(\log n)$

Finding Min/Max of a Sequence

Idaho State University | Computer Science

Algorithm:

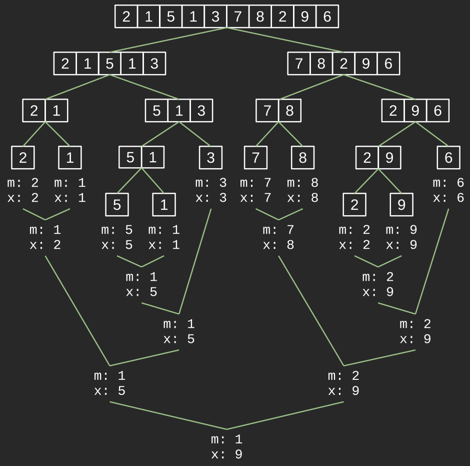


ROAR

Finding Min/Max of a Sequence

Idaho State University | Computer Science

Algorithm:



Recurrence Relation:

- Break the sequence into two subproblems (so $a = 2$) of size $\leq \lceil \frac{n}{2} \rceil$ (so $b = 2$)
- We then require two comparisons
  - One to compare the maxima
  - One to compare the minima

$$f(n) = 2f(n/2) + 2$$

pause

Complexity (from Master Theorem):

- Since the recurrence relation is $f(n) = 2f(n/2) + 2$ when n is even
- f is the number of comparisons required to find the minima and maxima of a sequence of size n
- By Theorem 1, the time complexity of $f(n)$ is $O(n^{\log 2}) = O(n)$

ROAR

Fast Multiplication of Integers

Algorithm:

- Suppose a and b are integers with binary expansions of length $2n$

$$
\begin{aligned}
a &= (a_{2n-1}a_{2n-2}\cdots a_1a_0)_2 \\
b &= (b_{2n-1}b_{2n-2}\cdots b_1b_0)_2
\end{aligned}
$$

- Let $a = 2^nA_1 + A_0$ and $b = 2^nB_1 + B_0$, where

$$
\begin{aligned}
A_1 &= (a_{2n-1}\cdots a_{n+1}bn)_2 \\
A_0 &= (a_{n-1}\cdots a_1a_0)_2 \\
B_1 &= (b_{2n-1}\cdots b_{n+1}bn)_2 \\
B_0 &= (b_{n-1}\cdots b_1b_0)_2
\end{aligned}
$$

- We can then rewrite ab as

$$ab = (2^{2n}+2^n)A_1B_1+2^n(A_1-A_0)(B_0-B_1)+(2^n+1)A_0B_0$$

ROAR

Fast Multiplication of Integers

Idaho State University | Computer Science

Algorithm:

- Suppose a and b are integers with binary expansions of length $2n$

$$\begin{aligned} a &= (a_{2n-1}a_{2n-2}\cdots a_1a_0)_2 \\ b &= (b_{2n-1}b_{2n-2}\cdots b_1b_0)_2 \end{aligned}$$

- Let $a = 2^nA_1 + A_0$ and $b = 2^nB_1 + B_0$, where

$$\begin{aligned} A_1 &= (a_{2n-1}\cdots a_{n+1}bn)_2 \\ A_0 &= (a_{n-1}\cdots a_1a_0)_2 \\ B_1 &= (b_{2n-1}\cdots b_{n+1}bn)_2 \\ B_0 &= (b_{n-1}\cdots b_1b_0)_2 \end{aligned}$$

- We can then rewrite ab as

$$ab = (2^{2n}+2^n)A_1B_1+2^n(A_1-A_0)(B_0-B_1)+(2^n+1)A_0B_0$$

Recurrence Relation:

- We split the multiplication of $2n$-bit integers into 3 multiplications ($a = 3$) of n-bit integers ($b = 2$), plus shifts and additions (a constant C)

$$f(n) = 3f(n/2) + Cn$$

Fast Multiplication of Integers

Algorithm:

- Suppose a and b are integers with binary expansions of length $2n$

$$\begin{aligned} a &= (a_{2n-1}a_{2n-2}\cdots a_1 a_0)_2 \\ b &= (b_{2n-1}b_{2n-2}\cdots b_1 b_0)_2 \end{aligned}$$

- Let $a = 2^n A_1 + A_0$ and $b = 2^n B_1 + B_0$, where

$$\begin{aligned} A_1 &= (a_{2n-1}\cdots a_{n+1} bn)_2 \\ A_0 &= (a_{n-1}\cdots a_1 a_0)_2 \\ B_1 &= (b_{2n-1}\cdots b_{n+1} bn)_2 \\ B_0 &= (b_{n-1}\cdots b_1 b_0)_2 \end{aligned}$$

- We can then rewrite ab as

$$ab = (2^{2n}+2^n)A_1B_1+2^n(A_1-A_0)(B_0-B_1)+(2^n+1)A_0B_0$$

Recurrence Relation:

- We split the multiplication of $2n$-bit integers into 3 multiplications ($a = 3$) of n-bit integers ($b = 2$), plus shifts and additions (a constant C)

$$f(n) = 3f(n/2) + Cn$$

Complexity (from Master Theorem):

- Since the recurrence relation is $f(n) = 3f(n/2) + Cn$, when n is even
- Where $f(n)$ is the number of bit operations needed to multiply to n-bit integers
- By the Master Theorem, $f(n)$ is $O(n^{\log 3})$ which is considerably faster than $O(n^2)$ of the conventional algorithm

ROAR

Mergesort

Algorithm:

procedure MSORT(L)
  if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L1 \leftarrow L_1, L_2, \ldots, L_m$
    $L2 := L_{m+1}, L_{m+2}, \ldots, L_n$
    $L :=$ MERGE(MSORT($L1$), MSORT($L2$))

procedure MERGE($L1, L2$)
  $L := [\,]$
  while $L1$ and $L2$ are both nonempty **do**
    remove smaller of $L1_1$ and $L2_1$, add to L
  if one list is empty **then**
    remove all elements of the other list and append to L
  return L

Mergesort

Algorithm:

procedure MSORT(L)
  if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L1 \leftarrow L_1, L_2, \ldots, L_m$
    $L2 := L_{m+1}, L_{m+2}, \ldots, L_n$
    $L :=$ MERGE(MSORT($L1$), MSORT($L2$))

procedure MERGE($L1, L2$)
  $L := [\,]$
  while $L1$ and $L2$ are both nonempty **do**
    remove smaller of $L1_1$ and $L2_1$, add to L
  if one list is empty **then**
    remove all elements of the other list and append to L
  return L

Recurrence Relation:

- Break list of length n into 2 sublists ($a = 2$), each of size $\leq \lceil \frac{n}{2} \rceil$ (so $b = 2$), then using $< n$ comparisons merge them. So

$$M(n) = 2M\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

Mergesort

Algorithm:

procedure MSORT(L)
  if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L1 \leftarrow L_1, L_2, \ldots, L_m$
    $L2 := L_{m+1}, L_{m+2}, \ldots, L_n$
    $L :=$ MERGE(MSORT($L1$), MSORT($L2$))

procedure MERGE($L1, L2$)
  $L := [\,]$
  while $L1$ and $L2$ are both nonempty **do**
    remove smaller of $L1_1$ and $L2_1$, add to L
  if one list is empty **then**
    remove all elements of the other list and append to L
  return L

Recurrence Relation:

- Break list of length n into 2 sublists ($a = 2$), each of size $\leq \lceil \frac{n}{2} \rceil$ (so $b = 2$), then using $< n$ comparisons merge them. So

$$M(n) = 2M\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

Complexity (from Master Theorem):

- Since the recurrence relation is $M(n) = 2M(n/2) + n$, when n is even
- Where $M(n)$ is the number of comparisons needed to sort n elements in a list
- By the Master Theorem, we find that $M(n)$ is $O(n \log n)$

ROAR

Are there any questions?