Lecture 11: Midterm Review

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

- Homework 2 deadline has passed
  - Solutions released on Canvas as of this morning
  - Will be graded by early next week, please be patient
    - Do not try to grade yourself in detail
    - Do not ask us “how many points will I get for…”
    - Just wait and you can ask for clarification/modification after grading is done!
- Midterm 1 to be released TONIGHT 8PM and due on Friday 8PM (Boston times)
  - Some review questions answered today!

This week

Today:

- First half-ish: Continue with graph algorithms
- Second half-ish: Answers to student-submitted questions
    - Form link sent out on Piazza

Thursday:

- No class while midterm exam is out

This week

Today:

- ~~First half-ish: Continue with graph algorithms~~
- ~~Second half-ish:~~ Answers to student-submitted questions
  - Form link sent out on Piazza

Thursday:

- No class while midterm exam is out

Midterm basic info

Reminder:

- Absolutely NO collaboration of any kind or use of the internet to find solutions is allowed
- You can use the Erickson book or the CLR book, as well as the lectures, slides, and any notes you have taken

Topics:

- Asymptotic order of growth
- Recurrence relations
- Proof by induction
- Recursive algorithms
- Dynamic programming

If you understand the solutions to the homework problems and have followed along with the lectures, you will do fine!

Midterm basic info

You should direct any midterm related questions to Piazza

- Ask all questions privately to all of the instructors
- If we think the answer is relevant for everyone, we will make it public or write it in a note.
- You should not publicly post anything about the exam anywhere, including on Piazza.

We will also be holding office hours as scheduled, however do not expect nearly as many hints as we give for the homework assignments!

- We are likely only going to answer clarifying questions
- Ask clarifying questions on Piazza BEFORE attending office hours
  - Often writing out your question helps you answer it yourself!

Format of Today’s Review

You have submit questions over the last few days via a Google Form

I have aggregated your questions and chosen some problems to go over

As we are going over the problems, you can ask me questions in the chat to help clarify

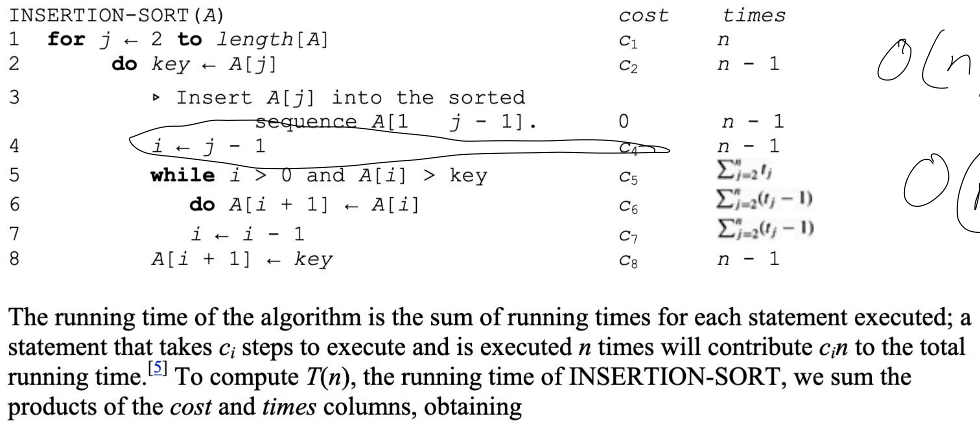
If we run in to a “Tim doesn’t have a good answer” type situation, I will post a Piazza note after lecture with a better answer

Constant time

When we say an operation takes “constant time”, we literally mean there is some constant c that does not depend on n that describes the number of instructions it takes to do that operation. We write it as $O(1)$.

Constant time

When we say an operation takes "constant time", we literally mean there is some constant c that does not depend on n that describes the number of instructions it takes to do that operation. We write it as $O(1)$.



INSERTION-SORT(*A*)

		cost	*times*
1	**for** $j \leftarrow 2$ **to** *length*[A]	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n - 1$
3	▹ Insert $A[j]$ into the sorted sequence $A[1 \quad j - 1]$.	0	$n - 1$
4	$i \leftarrow j - 1$	c_4	$n - 1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i \leftarrow i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i + 1] \leftarrow key$	c_8	$n - 1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and is executed n times will contribute $c_i n$ to the total running time.[5] To compute $T(n)$, the running time of INSERTION-SORT, we sum the products of the *cost* and *times* columns, obtaining

$O(1) = O(c_4)$

$O(n)$

$O(n \cdot w)$

https://web.ist.utl.pt/~fabio.ferreira/material/asa/clrs.pdf

“Recurrences”: what do words mean?

"Recurrences": what do words mean?

When we are describing a solution to a problem in terms of subproblems, we have a recursive relationship like:

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

$$SubsetSumS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1,t-X[i]) & otherwise \end{cases}$$

“Recurrences”: what do words mean?

When we are describing a solution to a problem in terms of subproblems, we have a recursive relationship like:

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

We can call this a “recursive specification”

$$SubsetSumS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \vee SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

"Recurrences": what do words mean?

When we are describing a solution to a problem in terms of subproblems, we have a recursive relationship like:

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

We can call this a "recursive specification"

$$SubsetSumS(i, t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1, t) & if\ t < X[i] \\ SubSum(i+1, t) \lor SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

When we describe the runtime of an algorithm, we have a recursive relationship like:

$$T(n) = T\left(\frac{n}{2}\right) + 3 \qquad\qquad T(n) = \sum_{i=0}^{n-1} T(i) + 3$$

“Recurrences”: what do words mean?

Can be written in Pseudocode

When we are describing a solution to a problem in terms of subproblems, we have a recursive relationship like:

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

$$SubsetSumS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

We can call this a “recursive specification”

When we describe the runtime of an algorithm, we have a recursive relationship like:

$$T(n) = T\left(\frac{n}{2}\right) + 3 \qquad T(n) = \sum_{i=0}^{n-1} T(i) + 3$$

This is a “recurrence relation”

Running time of recursive algorithms

To get the running time of a recursive algorithm, we write a recurrence relation describing the time to run the algorithm on an input of size n in terms of the running time on inputs smaller than n.

Running time of recursive algorithms

To get the running time of a recursive algorithm, we write a recurrence relation describing the time to run the algorithm on an input of size n in terms of the running time on inputs smaller than n.

Let’s look back at how we wrote the recurrence relations for binary search and merge sort.

Binary Search Recurrence Relation

What does the recurrence relation look like for binary search?

```
StartSearch(A,t):
  // A[1:n] sorted in ascending order
  Return Search(A,1,n,t)

Search(A,$\ell$,r,t):
  If($\ell$ > r): return FALSE        $O(1)$

  m ← $\ell + \left\lfloor \frac{r-\ell}{2} \right\rfloor$   $O(1)$

  If(A[m] = t): return m
  ElseIf(A[m] > t): return Search(A,$\ell$,m-1,t)
  Else: return Search(A,m+1,r,t)
```

$$T(n) = T(\text{division}) + T(\text{conquer})$$

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$= T\left(\frac{n}{2}\right) + 3$$

MergeSort: Runtime Analysis

```
MERGESORT($A[1..n]$):
    if $n > 1$
        $m \leftarrow \lfloor n/2 \rfloor$
        MERGESORT($A[1..m]$)        ⟨⟨Recurse!⟩⟩
        MERGESORT($A[m+1..n]$)      ⟨⟨Recurse!⟩⟩
        MERGE($A[1..n], m$)
```

```
MERGE($A[1..n], m$):
    $i \leftarrow 1$; $j \leftarrow m+1$
    for $k \leftarrow 1$ to $n$
        if $j > n$
            $B[k] \leftarrow A[i]$; $i \leftarrow i+1$
        else if $i > m$
            $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
        else if $A[i] < A[j]$
            $B[k] \leftarrow A[i]$; $i \leftarrow i+1$
        else
            $B[k] \leftarrow A[j]$; $j \leftarrow j+1$

    for $k \leftarrow 1$ to $n$
        $A[k] \leftarrow B[k]$
```

$2n = O(n)$

Let's write down a *recurrence relation* that describes the runtime:

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

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

Difference between the two

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

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

Solving Recurrence Relations

3 Methods:

1. Master theorem (if applicable)
2. Writing a few values → guess and check
3. Recursion Trees

Master Theorem

- Recipe for recurrences of the form:
  - $T(n) = \boldsymbol{a} \cdot T(n/\boldsymbol{b}) + Cn^{\boldsymbol{d}}$
- Three cases:
  - $\left(\frac{\boldsymbol{a}}{\boldsymbol{b}^{d}}\right) > 1 :\ T(n) = \Theta\left(n^{\log_{\boldsymbol{b}} \boldsymbol{a}}\right)$
  - $\left(\frac{\boldsymbol{a}}{\boldsymbol{b}^{d}}\right) = 1 :\ T(n) = \Theta\left(n^{\boldsymbol{d}} \log n\right)$
  - $\left(\frac{\boldsymbol{a}}{\boldsymbol{b}^{d}}\right) < 1 :\ T(n) = \Theta\left(n^{\boldsymbol{d}}\right)$

Note that the theorem does not apply to our MOMSelect recurrence:

$$\mathrm{T(n)} = \mathrm{T}(\tfrac{7n}{10}) + \mathrm{T}(\tfrac{n}{5}) + \mathrm{O(n)}$$

Binary Search:

$\mathrm{T(n)} = \mathrm{T}(\frac{n}{2}) + \mathrm{O}(1)$

$\mathrm{T(n)} = 1\mathrm{T}(\frac{n}{2}) + \mathrm{n}^0$

$$\left(\frac{1}{2^0}\right) = 1$$

So

$$T(n) = \Theta\left(n^{\mathbf{0}} \log n\right)$$

and we get

$$T(n) = \Theta(\log n)$$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

$$T(1) = 2T(0) + 1 = 1$$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

$$T(1) = 2T(0) + 1 = 1$$

$$T(2) = 2T(1) + 1 = 2 + 1 = 3$$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

$$T(1) = 2T(0) + 1 = 1$$

$$T(2) = 2T(1) + 1 = 2 + 1 = 3$$

$$T(3) = 2 \cdot 3 + 1 = 7$$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

$$T(1) = 2T(0) + 1 = 1$$

$$T(2) = 2T(1) + 1 = 2 + 1 = 3$$

There is a pattern here!

$$T(3) = 2 \cdot 3 + 1 = 7$$

$$T(4) = 2 \cdot 7 + 1 = 15$$

$$T(5) = 2 \cdot 15 + 1 = 31$$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

$$T(1) = 2T(0) + 1 = 1 = 2^1 - 1$$

$$T(2) = 2T(1) + 1 = 2 + 1 = 3 = 2^2 - 1$$

$$T(3) = 2 \cdot 3 + 1 = 7 = 2^3 - 1$$

$$T(4) = 2 \cdot 7 + 1 = 15 = 2^4 - 1$$

$$T(5) = 2 \cdot 15 + 1 = 31 = 2^5 - 1$$

There is a pattern here!

We can use it to "guess" the running time.

What is our guess?

$2^n - 1$

Solving $T(n) = 2T(n-1) + 1$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

$$T(1) = 2T(0) + 1 = 1$$

$$T(2) = 2T(1) + 1 = 2 + 1 = 3$$

$$T(3) = 2 \cdot 3 + 1 = 7$$

$$T(4) = 2 \cdot 7 + 1 = 15$$

$$T(5) = 2 \cdot 15 + 1 = 31$$

There is a pattern here!

We can use it to "guess" the running time.

What is our guess?

Now we need to prove it!

Proving $T(n) = 2T(n-1) + 1 = O(2^n)$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

Proving $T(n) = 2T(n-1) + 1 = O(2^n)$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

Assume for induction that $T(n) = 2T(n-1) + 1 \leq c2^n + 1$ (for some constant c)

We will show that $T(n+1) = 2T(n+1-1) + 1 \leq c2^{n+1} + 1$

Proving $T(n) = 2T(n-1) + 1 = O(2^n)$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

Assume for induction that $T(n) = 2T(n-1) + 1 \leq c2^n + 1$ (for some constant c)

We will show that $T(n+1) = 2T(n+1-1) + 1 \leq c2^{n+1} + 1$

We have $2T(n) \leq c2^{n+1}$

Proving $T(n) = 2T(n-1) + 1 = O(2^n)$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

Assume for induction that $T(n) = 2T(n-1) + 1 \leq c2^n + 1$ (for some constant c)

We will show that $T(n+1) = 2T(n+1-1) + 1 \leq c2^{n+1} + 1$

We have
$$2T(n) \leq c2^{n+1}$$
$$2 \cdot 2^n c \leq c2^{n+1}$$
By the inductive hypothesis

Proving $T(n) = 2T(n-1) + 1 = O(2^n)$

$$T(n) = 2T(n-1) + 1,$$
$$T(0) = 0$$

Assume for induction that $T(n) = 2T(n-1) + 1 \leq c2^n + 1$ (for some constant c)

We will show that $T(n+1) = 2T(n+1-1) + 1 \leq c2^{n+1} + 1$

We have
$$2T(n) \leq c2^{n+1}$$
$$2 \cdot 2^n c \leq c2^{n+1}$$ By the inductive hypothesis
$$2^{n+1} \leq 2^{n+1}$$
$$2^{n+1} + 1 = O(2^{n+1})$$

Recursion Tree: MOMSelect Running Time

```
MOMSelect(A[1..n], k):
  If n <= 25:
    return median(A)
  Else:
    mom ← MOM(A[1..n])
    r ← Partition(A, mom)

    If k < r:
      Return MOMSelect(A[1..r], k)
    ElseIf k > r:
      Return MOMSelect(A[r+1..n], k-r)
    Else:
      Return A[r]
```

What is a recurrence relation for MOMSelect?

T(n) = T(Selection) + T(MOM) + f(ops per step)

$$T(n) = T\left(\frac{7n}{10}\right) + T\left(\frac{3n}{10}\right) + O(n)$$

Recursion Tree

$$\overset{f(n)}{}$$

$T(n) = T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$ $= O(n)$

$T(n) = O(n)$

$$\begin{tikzcd}
 & & & \boxed{n} \arrow[dll] \arrow[drr] & & & \\
 & \boxed{\frac{n}{5}} \arrow[dl] \arrow[dr] & & + & & \boxed{\frac{7n}{10}} \arrow[dl] \arrow[dr] & \\
\boxed{\frac{n}{25}} & + & \boxed{\frac{7n}{50}} & + & \boxed{\frac{7n}{50}} & + & \boxed{\frac{49n}{100}}
\end{tikzcd}$$

$\frac{9}{10} n$

$\frac{4n + 28n + 49n}{100} = \frac{81}{100} n$

$= \left(\frac{9}{10}\right)^2 n$

Since the work at each level is decreasing exponentially, the O(n) term dominates!

T(n) = T($\frac{7n}{10}$) + T($\frac{n}{5}$) + O(n)
T(1) = 1

Proof by induction

We want to show that

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \leq O(n),$$ meaning

T(n) = T($\frac{7n}{10}$) + T($\frac{n}{5}$) + O(n)
T(1) = 1

Proof by induction

We want to show that

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \leq O(n),$$ meaning

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + n \leq Cn$$ (for some C)

Proof by induction

T(n) = T($\frac{7n}{10}$) + T($\frac{n}{5}$) + O(n)
T(1) = 1

We want to show that

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \leq O(n),$$ meaning

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + n \leq Cn$$ (for some C)

By induction, since $\frac{1}{5}n < \frac{7}{10}n < n$, we have

$$C\frac{7n}{10} + C\frac{n}{5} + n$$

Proof by induction

T(n) = T($\frac{7n}{10}$) + T($\frac{n}{5}$) + O(n)
T(1) = 1

We want to show that

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \leq O(n), \text{ meaning}$$

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + n \leq Cn \text{ (for some } C\text{)}$$

By induction, since $\frac{1}{5}n < \frac{7}{10}n < n$, we have

$$C\frac{7n}{10} + C\frac{n}{5} + n$$

Pulling out n, we get

$$n\left(C\frac{7}{10} + C\frac{1}{5} + 1\right)$$

$$n\left(C\frac{9}{10} + 1\right)$$

$$\leq Cn$$

Proof by induction

T(n) = T($\frac{7n}{10}$) + T($\frac{n}{5}$) + O(n)
T(1) = 1

We want to show that

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \leq O(n),$$ meaning

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + n \leq Cn$$ (for some C)

By induction, since $\frac{1}{5}n < \frac{7}{10}n < n$, we have

$$C\frac{7n}{10} + C\frac{n}{5} + n$$

Pulling out n, we get

$$n\left(C\frac{7}{10} + C\frac{1}{5} + 1\right)$$

$$n\left(C\frac{9}{10} + 1\right)$$

$$\leq Cn$$

For which values of C?

$$C\frac{9}{10} + 1 \leq C$$
$$9C + 10 \leq 10C$$
$$C \geq 10$$

T(n) = T($\frac{7n}{10}$) + T($\frac{n}{5}$) + O(n)
T(1) = 1

Proof by induction

We want to show that

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \leq O(n),$$ meaning

$$T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + n \leq Cn$$ (for some C)

By induction, since $\frac{1}{5}n < \frac{7}{10}n < n$, we have

$$C\frac{7n}{10} + C\frac{n}{5} + n$$

Pulling out n, we get

$$n\left(C\frac{7}{10} + C\frac{1}{5} + 1\right)$$

$$n\left(C\frac{9}{10} + 1\right)$$

$$\leq Cn$$ (as long as $C \geq 10$)

For which values of C?

$$C\frac{9}{10} + 1 \leq C$$
$$9C + 10 \leq 10C$$
$$C \geq 10$$

Reporting running times: tight vs. loose bounds

- Usually in this class it should be pretty clear which bounds are appropriate from context

- For recursive algorithms, if you write down a recurrence relation, you should be able to find a bound for it

- If you think there is a tighter bound than the one you've written, you should try to prove it
    - But you don't want to be doing original research, so keep it simple!

Backtracking

- So far, we have seen cases where the next recursive call is clear
    - In MergeSort, we need both left and right subarrays to be sorted
    - In MOMSelect and BinarySearch, we guarantee the value we are looking for is in a specific subarray
- What if we can’t tell which decision to make?
- Enter *backtracking*: When we need to make a decision, try one small step in all directions and evaluate all outcomes.

N Queens

Problem statement: Given an *n*x*n* dimensional chessboard, place *n* queens on the board such that none can attack each other.

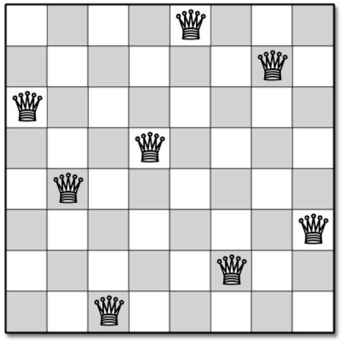


Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array $[5,7,1,4,2,8,6,3]$

Idea: Incrementally build a solution by placing one queen at a time!

Given an arbitrary *n*, how can we decide where to place queens?

N Queens

Idea: Incrementally build a solution by placing one queen at a time!

```
PlaceQueens(Q[1..n], r):
  If r = n+1:
    print Q[1..n]
  Else:
    for j ← 1 to n:
      legal ← True
      for i ← 1 to r − 1:
        if(Q[i]=j) or
          (Q[i]=j+r-i) or
          (Q[i] = j − r):
            legal ← False
      if legal:
        Q[r] ← j
        PlaceQueens(Q[1..n], r+1)
```

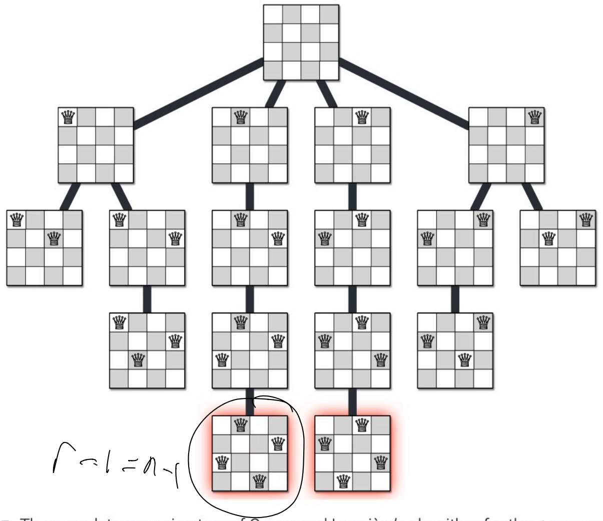


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

N Queens

Idea: Incrementally build a solution by placing one queen at a time!

```
PlaceQueens(Q[1..n], r):
  If r = n+1:
    print Q[1..n]
  Else:
    for j ← 1 to n:
      ~~Q[r] = j~~  Q[r+1] = j
      if CheckLegal(Q[1..r]):
        PlaceQueens(Q[1..n], r+1)
```

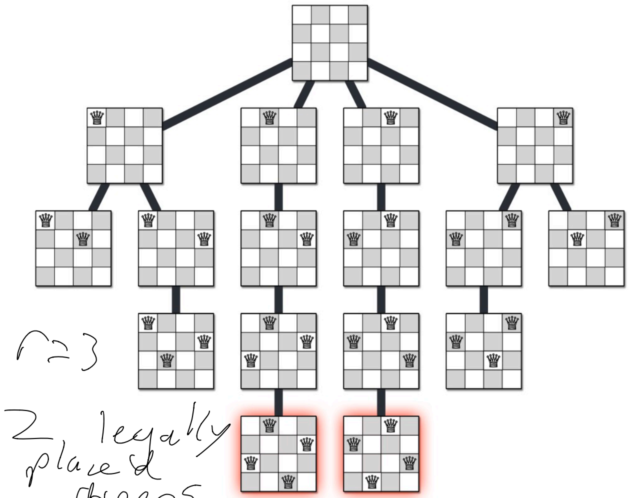


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

Backtracking pattern

Idea: Incrementally build a solution by placing one queen at a time!

- Appropriate when a sequence of incremental decisions can enumerate solutions
  - Solution is often itself a sequence, e.g. Q[1..n] is a sequence of queens placed in rows 1..n
- Exactly 1 decision is made at every step
  - We usually need some information about previous decisions, but this should be as small as possible
- Problem is solved by *recursive brute force*, meaning we do not "prune" decisions that are obviously bad (leaves in the tree)

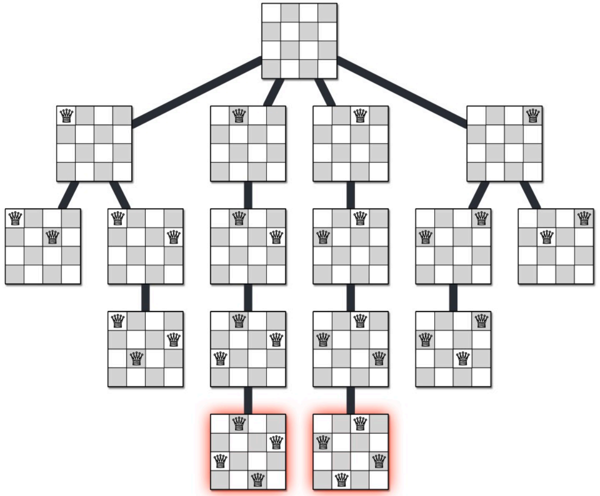


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

Proof of correctness for dynamic programming

We prove a dynamic programming algorithm is correct by showing that the recursive specification is correct by induction.

- If it is easier for you to think through, you can write the *recursive* algorithm in pseudocode and prove it from there (since they are equivalent)

If we know the recursive specification is correct, then it is straightforward to explain that the iterative algorithm is also correct, since all it does is realize the recursive specification!

General Notes and Strategies

- I am not trying to trick you, some questions have simple answers
- Recall what we have already done and start thinking from there
- Make sure you understand the problem before you start working on the solution
- If you are feeling frustrated, ask a question and/or take a break!
- Please sleep. You will not do better if you pull an all-nighter and there are no points for “most time spent staring at blank page”

Wrap Up

Good luck!