CSE373: Data Structure & Algorithms

Lecture 21: More Sorting and Other Classes of Algorithms

Lauren Milne

Summer 2015

Admin

Homework 5 due tonight!

Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

Simple algorithms: $O(n^2)$	**Fancier algorithms: $O(n \log n)$**	**Comparison lower bound: $\Omega(n \log n)$**	**Specialized algorithms: $O(n)$**	**Handling huge data sets**
Insertion sort	**Heap sort**		**Bucket sort**	**External sorting**
Selection sort	**Merge sort**		**Radix sort**	
Shell sort	**Quick sort**			
…	…			

Bucket Sort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and *K* (or any small range):
  - Create an array of size *K*
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

`count` array	
1	3
2	1
3	2
4	2
5	3

- Example:

  K=5

  input (5,1,3,4,3,2,1,1,5,4,5)

  output: 1,1,1,2,3,3,4,4,5,5,5

Analyzing Bucket Sort

- Overall: $O(n+K)$
  - Linear in n, but also linear in K
  - $\Omega(n \log n)$ lower bound does not apply because this is not a comparison sort

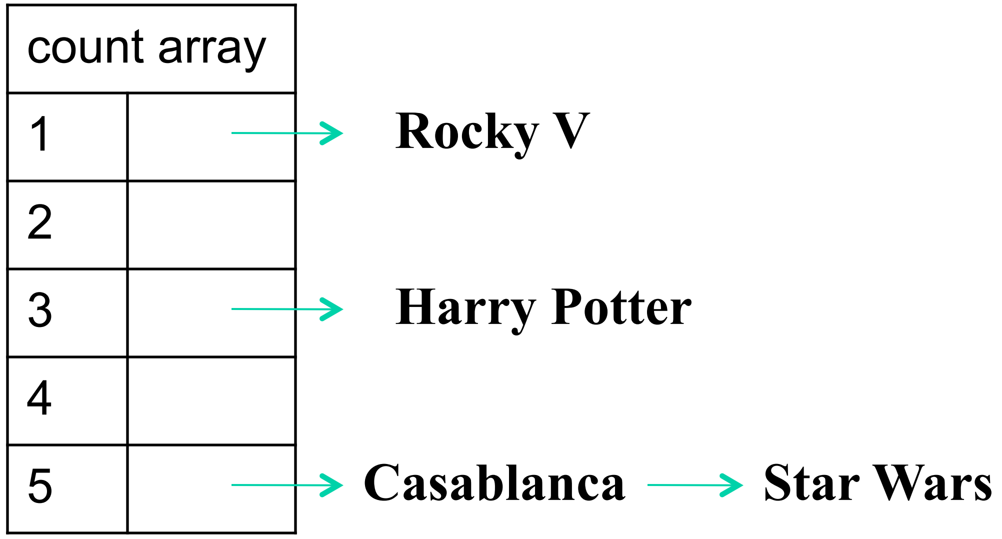
- Good when K is smaller (or not much larger) than n
  - We don’t spend time doing comparisons of duplicates

- Bad when K is much larger than n
  - Wasted space; wasted time during linear $O(K)$ pass

- For data in addition to integer keys, use list at each bucket

Bucket Sort with Data

- Most real lists aren't just keys; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, insert in *O*(1) (at beginning, or keep pointer to last element)



- Example: Movie ratings; scale 1-5;1=bad, 5=excellent

  Input=

  5: Casablanca

  3: Harry Potter movies

  5: Star Wars Original Trilogy

  1: Rocky V

- Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
- Easy to keep 'stable'; Casablanca still before Star Wars

Visualization

- http://www.cs.usfca.edu/~galles/visualization/CountingSort.html

Radix sort

- Origins go back to the 1890 U.S. census
- Radix = “the base of a number system”
  - Examples will use 10 because we are used to that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128

- Idea:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least* significant digit
    - Keeping sort *stable*
  - Do one pass per digit
  - Invariant: After *k* passes (digits), the last *k* digits are sorted

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

Input:
478
537
9
721
3
38
143
67

First pass:

bucket sort by ones digit

Order now:
721
3
143
537
67
478
38
9

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Order was: 721
3
143
537
67
478
38
9

Second pass:

stable bucket sort by tens digit

Order now: 3
9
721
537
38
143
67
478

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

0	1	2	3	4	5	6	7	8	9
3 9 38 67	143			478	537		721		

Order was:

3
9
721
537
38
143
67
478

Third pass:

stable bucket sort by 100s digit

Order now:

3
9
38
67
143
478
537
721

Analysis

Input size: n

Number of buckets = Radix: B

Number of passes = “Digits”: P

Work per pass is 1 bucket sort: $O(B+n)$

Total work is $O(P(B+n))$

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
    - Run-time proportional to: $15*(52 + n)$
    - This is less than n log n only if $n > 33{,}000$
    - Of course, cross-over point depends on constant factors of the implementations
        - And radix sort can have poor locality properties

Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

Simple algorithms: $O(n^2)$	**Fancier algorithms: $O(n \log n)$**	**Comparison lower bound: $\Omega(n \log n)$**	**Specialized algorithms: $O(n)$**	**Handling huge data sets**
Insertion sort	**Heap sort**		**Bucket sort**	**External sorting**
Selection sort	**Merge sort**		**Radix sort**	
Shell sort	**Quick sort**			
…	…			

Sorting massive data

- Need sorting algorithms that minimize disk/tape access time:
  - Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
  - Merge sort scans linearly through arrays, leading to (relatively) efficient sequential disk access

- Merge sort is the basis of massive sorting

- Merge sort can leverage multiple disks

External Merge Sort

- Sort 900 MB using 100 MB RAM
  - Read 100 MB of data into memory
  - Sort using conventional method (e.g. quicksort)
  - Write sorted 100MB to temp file
  - Repeat until all data in sorted chunks (900/100 = 9 total)
- Read first 10 MB of each sorted chuck, merge into remaining 10MB
  - writing and reading as necessary
  - Single merge pass instead of *log n*
  - Additional pass helpful if data much larger than memory
- Parallelism and better hardware can improve performance
- Distribution sorts (similar to bucket sort) are also used

Last Slide on Sorting

- Simple $O(n^2)$ sorts can be fastest for small n
  - Selection sort, Insertion sort (latter linear for mostly-sorted)
  - Good for “below a cut-off” to help divide-and-conquer sorts
- $O(n \texttt{ log } n)$ sorts
  - Heap sort, in-place but not stable nor parallelizable
  - Merge sort, not in place but stable and works as external sort
  - Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies
- $\Omega\ (n \texttt{ log } n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of possible key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!

Done with sorting! (phew..)

- Moving on….

- There are many many algorithm techniques in the world
  - We've learned a few

- What are a few other "classic" algorithm techniques you should at least have heard of?
  - And what are the main ideas behind how they work?

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A clever form of exhaustive search

Dynamic Programming: Idea

- Divide a bigger problem into many smaller subproblems
- If the number of subproblems grows exponentially, a recursive solution may have an exponential running time ☹
- Dynamic programming to the rescue! ☺
- Often an individual subproblem occurs many times!
  - Store the results of subproblems in a table and re-use them instead of recomputing them
  - Technique called memoization

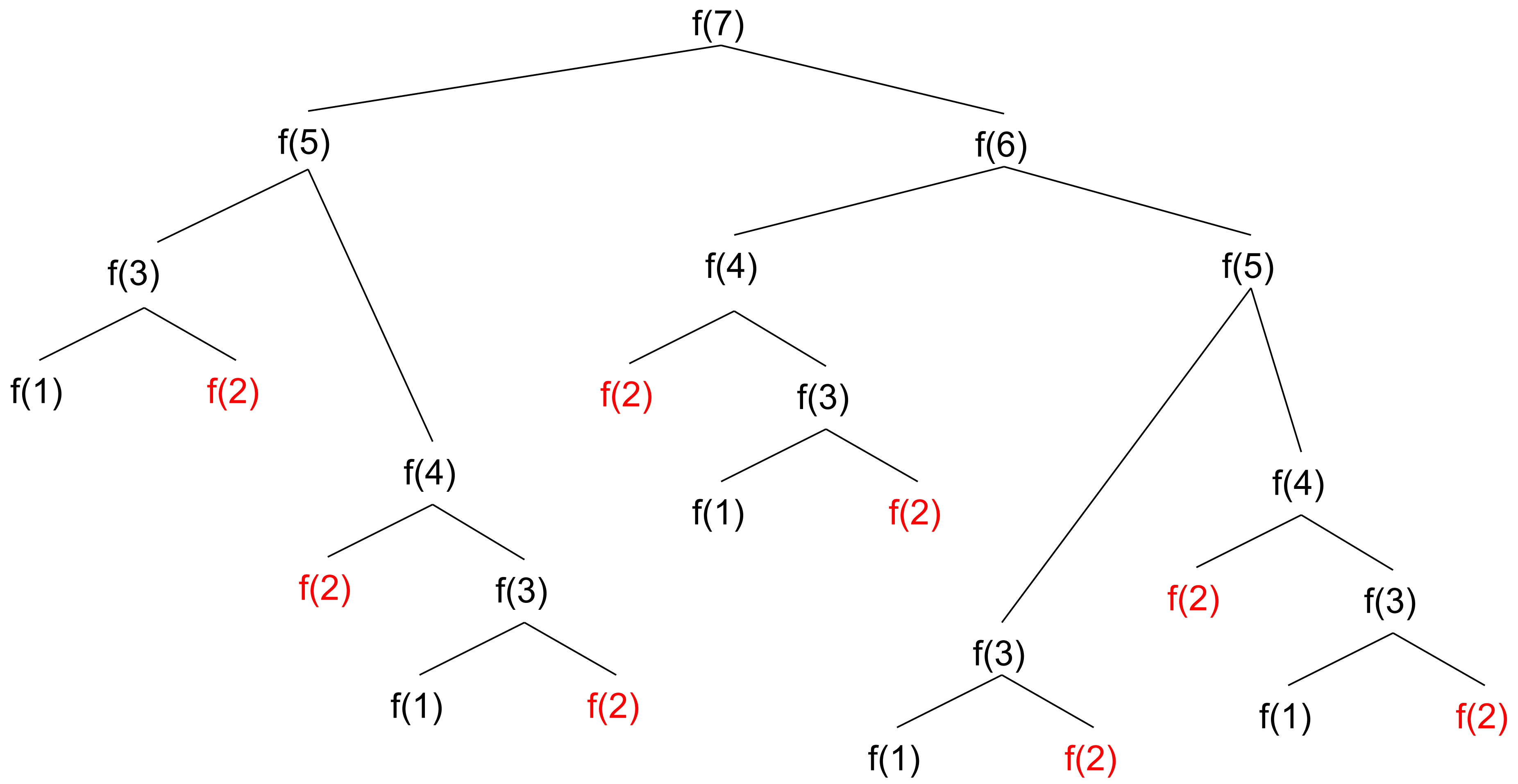
Fibonacci Sequence: Recursive

- The fibonacci sequence is a very famous number sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- The next number is found by adding up the two numbers before it.
- Recursive solution:

```
fib(int n) {
  if (n == 1 || n == 2) {
    return 1
  }
  return fib(n - 2) + fib(n - 1)
}
```

- Exponential running time!
  - A lot of repeated computation

Repeated computation



Fibonacci Sequence: memoized

```
fib(int n) {
  Map results = new Map()
  results.put(1, 1)
  results.put(2, 1)
  return fibHelper(n, results)
}
fibHelper(int n, Map results) {
  if (!results.contains(n)) {
    results.put(n, fibHelper(n-2)+fibHelper(n-1))
  }
  return results.get(n)
}
```

Now each call of **`fib(x)`** only gets computed once for each x!

Dynamic Programming

- Work "from the bottom up" & save the results of simpler problems
  - solutions to simpler problems are used to compute the solution to more complex problems

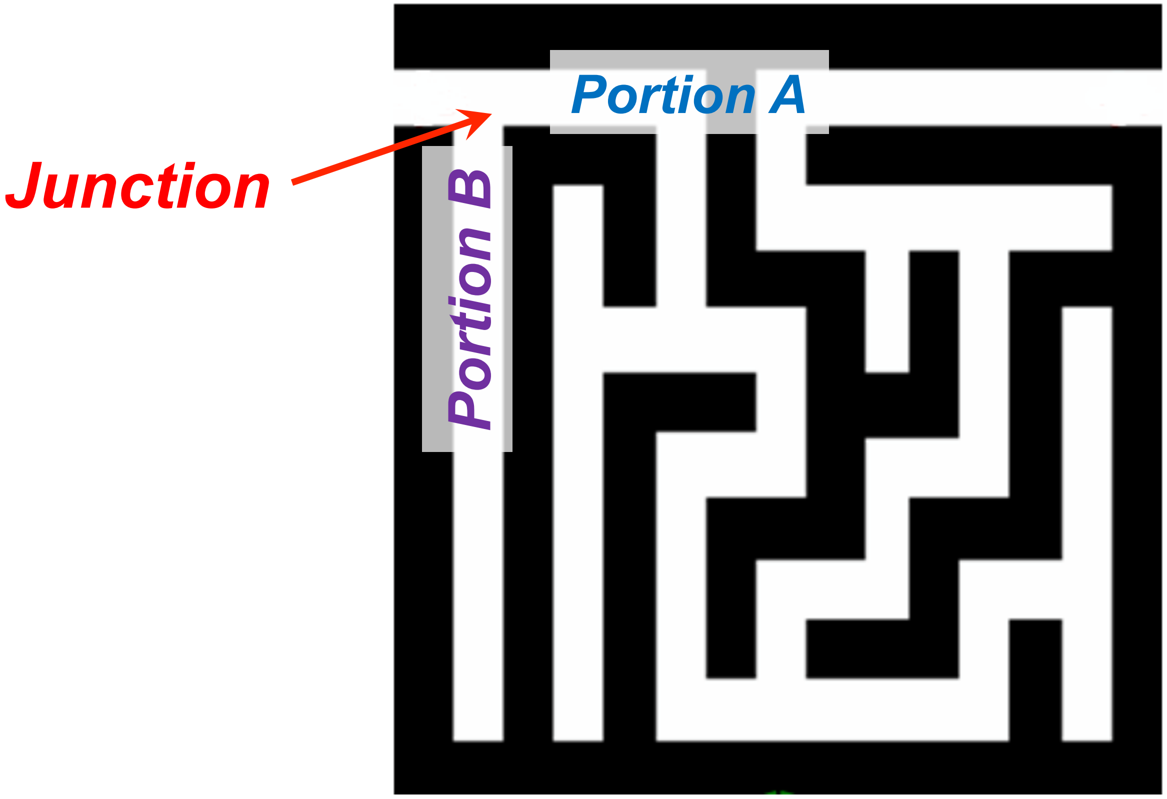
- Used for optimization problems, especially ones that would otherwise take exponential time
  - Must satisfy the principle of optimality i.e. the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A clever form of exhaustive search

Backtracking: Idea

- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
  - At some point, you might have two options of which direction to go:



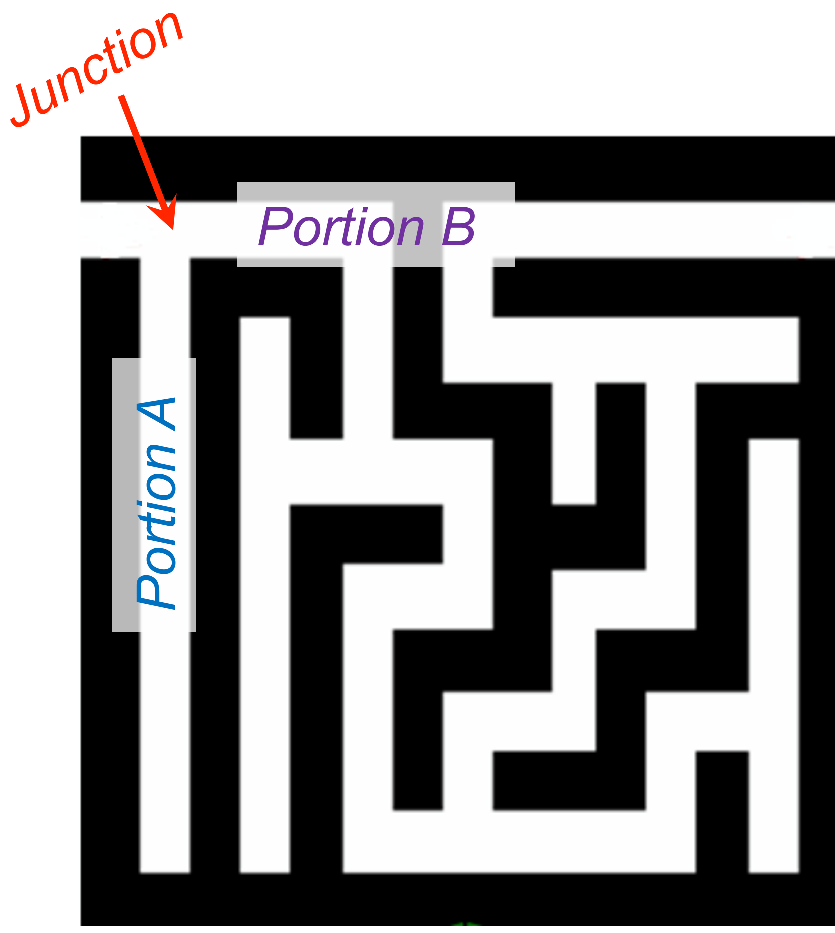
Backtracking

One strategy would be to try going through Portion A of the maze.

If you get stuck before you find your way out, then you *"backtrack"* to the junction.

At this point in time you know that Portion A will *NOT* lead you out of the maze,

so you then start searching in Portion B

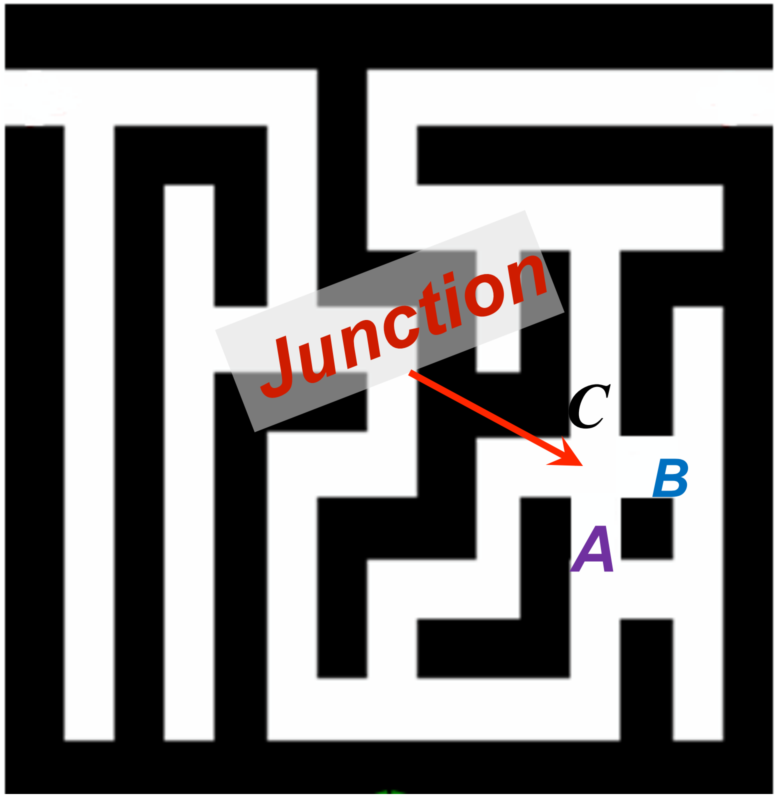


Backtracking

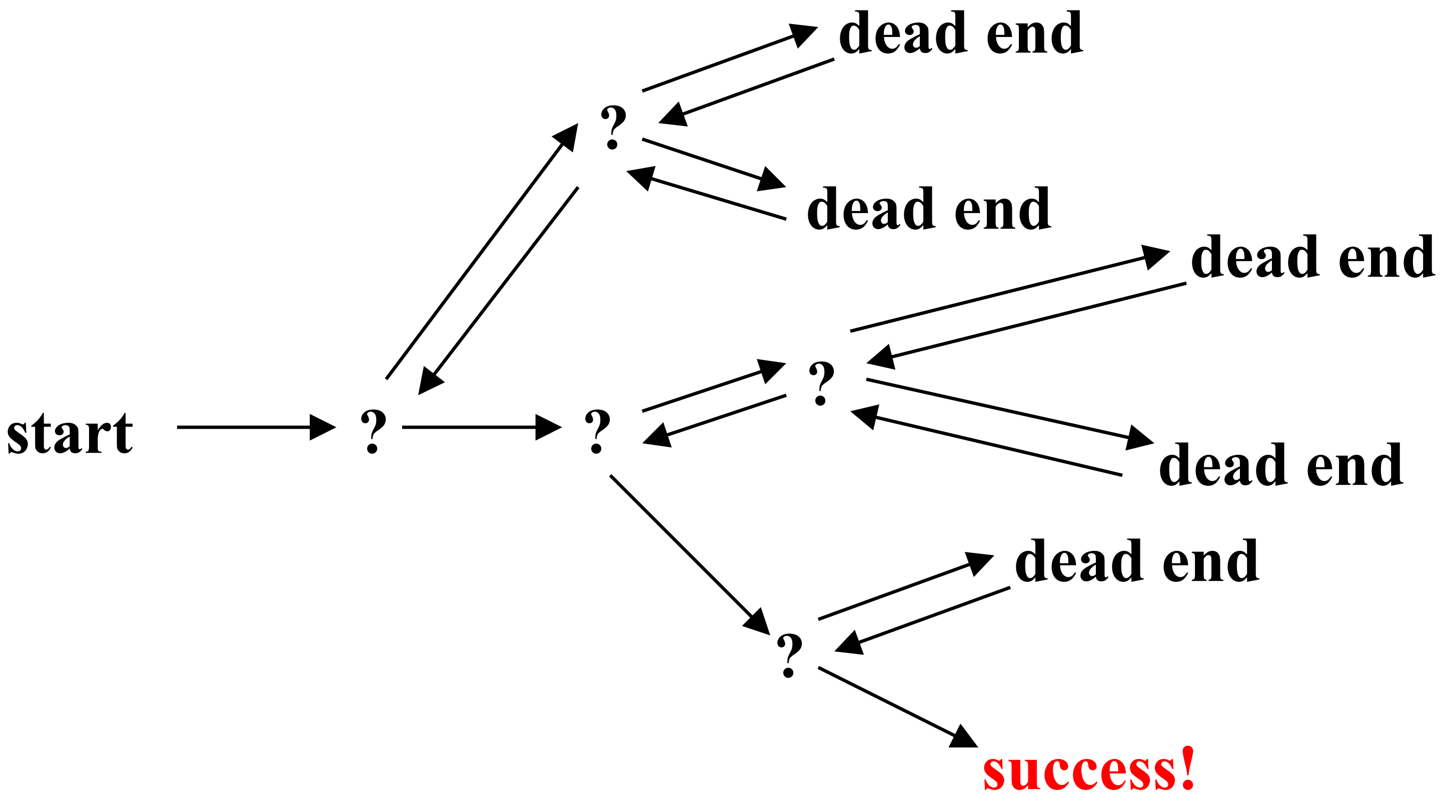
- Clearly, at a single junction you could have even more than 2 choices.

- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, *"backtrack"* to the junction and try the next choice.

- If you try all choices and never found a way out, then there IS no solution to the maze.



Backtracking (animation)



Backtracking

- Dealing with the maze:
  - From your start point, you will iterate through each possible starting move.
  - From there, you recursively move forward.
  - If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.

- Make sure you don't try too many possibilities,
  - Mark which locations in the maze have been visited already so that no location in the maze gets visited twice.
  - If a place has already been visited, there is no point in trying to reach the end of the maze from there again.

Backtracking

The neat thing about coding up backtracking is that it can be done recursively, without having to do all the bookkeeping at once.

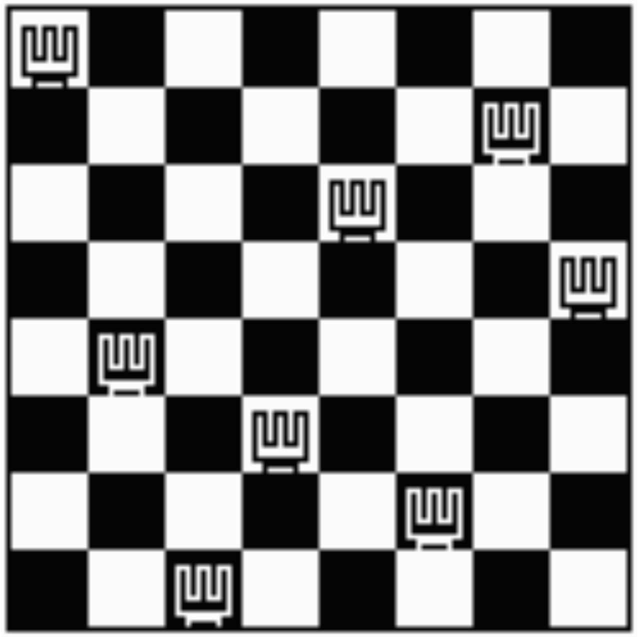
- Instead, the stack of recursive calls does most of the bookkeeping
- (i.e., keeps track of which locations we’ve tried so far.)

Backtracking: The 8 queens problem

- Find an arrangement of **8** queens on a single chess board such that no two queens are attacking one another.

- In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).

  – Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.

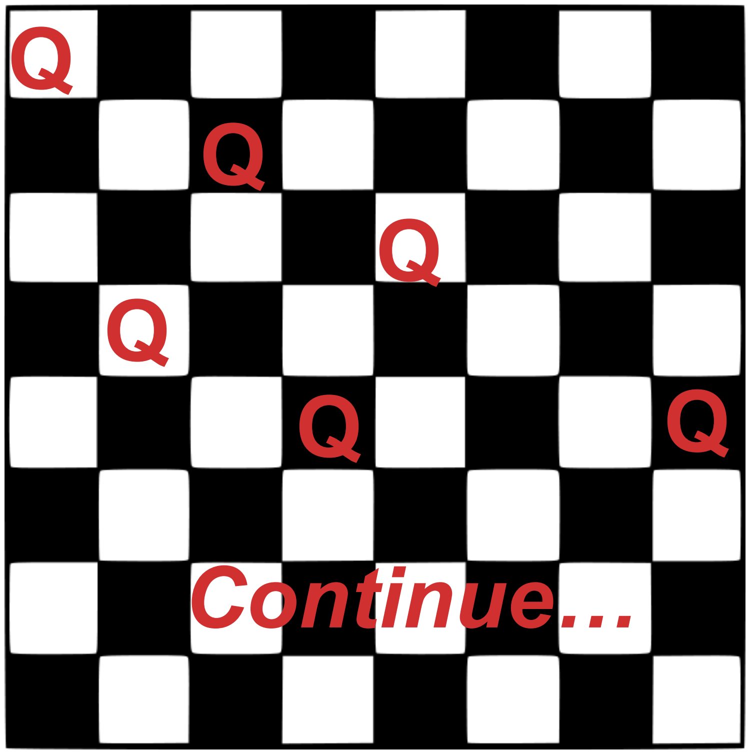


Backtracking

The backtracking strategy is as follows:

1) Place a queen on the first available square in row 1.
2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3) Continue in this fashion until either:
   a) You have solved the problem, or
   b) You get stuck.

   When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.



Animated Example:
http://www.hbmeyer.de/backtrack/achtdamen/eight.htm#up

Backtracking – 8 queens Analysis

- Another possible brute-force algorithm is generate all possible permutations of the numbers 1 through 8 (there are 8! = 40,320),
  - Use the elements of each permutation as possible positions in which to place a queen on each row.
  - Reject those boards with diagonal attacking positions.

- The backtracking algorithm does a bit better
  - constructs the search tree by considering one row of the board at a time, eliminating most non-solution board positions at a very early stage in their construction.
  - because it rejects row and diagonal attacks even on incomplete boards, it examines only 15,720 possible queen placements.

- 15,720 is still a lot of possibilities to consider
  - Sometimes we have no other choice but to do the best we can ☺

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A clever form of exhaustive search