

COMP 321 – Programming Challenges

Algorithm Concepts #1 – Roadmap of Search, Greedy and DP

Sept and 27, 2019

Outline for today

- Check-in on Assignment 1
- The most common 3 code structures: search, greedy, DP
 - How and when to select, and broad definitions
- Walking through practical examples that can come up on contests

The Contest Question Roadmap

Try enumerating
all possibilities

TLE?

Can pruning make
it fast enough?

TLE?

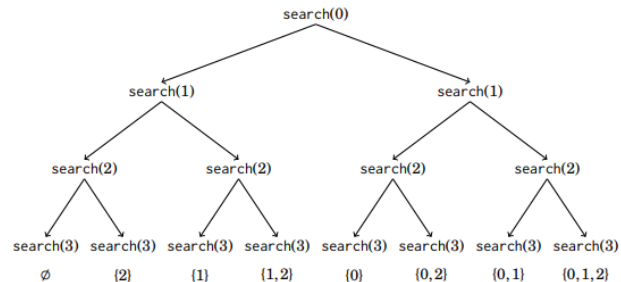
Find a correct Greedy
choice at each step?

WA?

Need to look for a
divide-and-conquer
recurrence

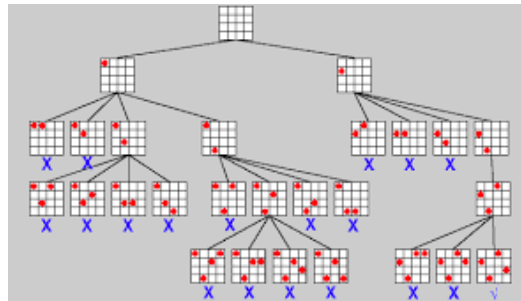
AC?

Brute
Force!



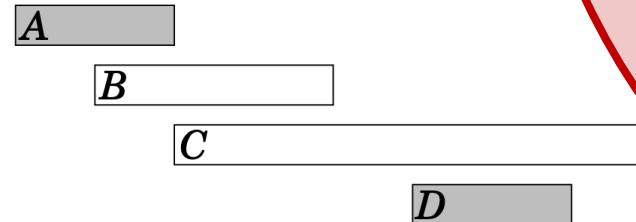
AC?

Pruned
Search!



AC?

Greedy!



Next lecture

Bottom-up DP

COINS	C1	C2	C3		F[i]
i	AMOUNT				
1	F(0)	-	-		1
2	F(1)	F(0)	-		1
3	F(2)	F(1)	F(0)	min+1	1
4	F(3)	F(2)	F(1)		2
5	F(4)	F(3)	F(2)		2
6	F(5)	F(4)	F(3)		2

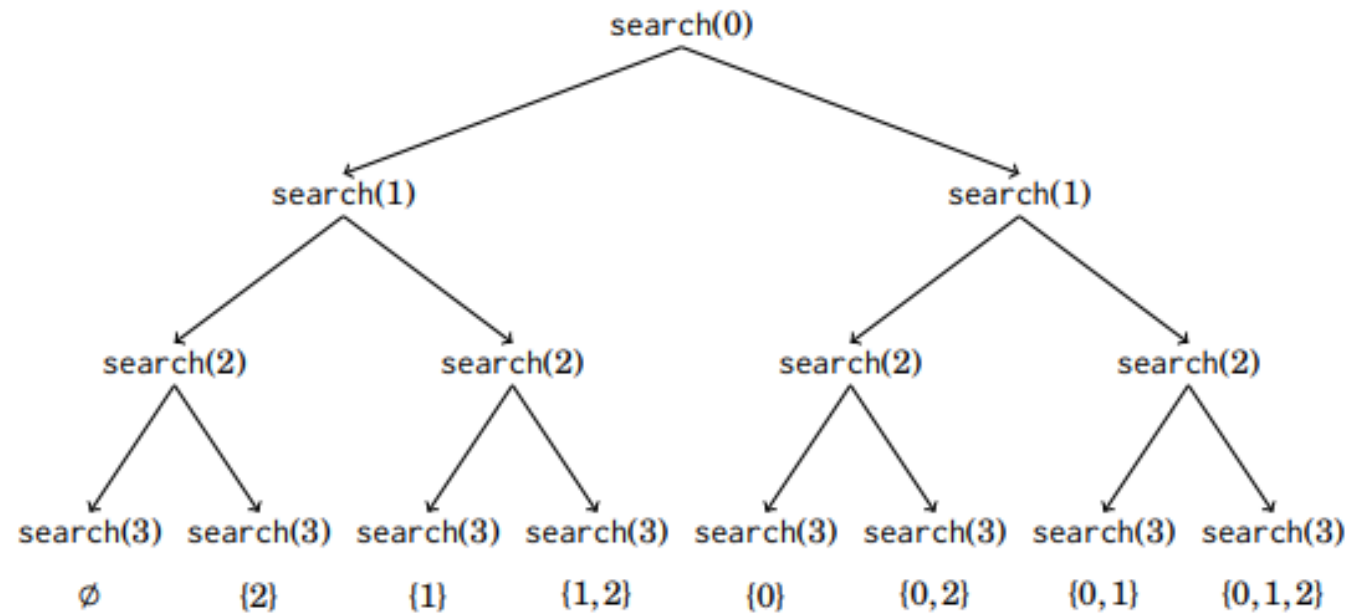
Finding minimal number of coins for amount $i = 6$.
 $F[6] = 2$

Top-down DP



Recall: code structure for brute-force (all-subset computation shown)

```
void search(int k) {  
    if (k == n) {  
        // process subset  
    } else {  
        search(k+1);  
        subset.push_back(k);  
        search(k+1);  
        subset.pop_back();  
    }  
}
```



From: Competitive Programmer's Handbook by Antti Laaksonen.

Of course, brute force will become too slow

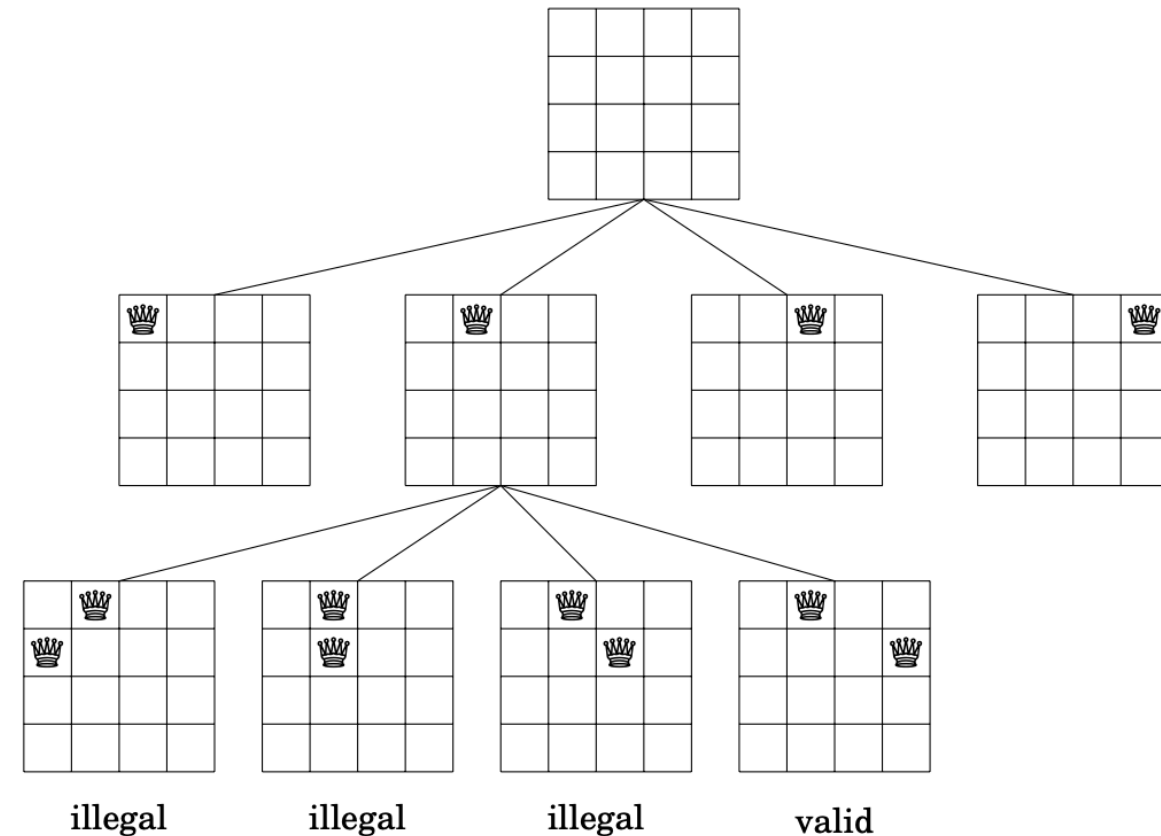
- Of course this can TLE even at pretty small limits:
 - $\binom{n}{r} = \frac{n!}{r!(n-r)!}$ ways to choose r out of n elements
 - Around 100K at $n=20$
 - 2^n sub-sets when $|S| = n$ (length is n)
 - Around 1M at $n=20$
 - $n!$ permutations of n elements
 - Around 10^{18} at $n=20$
- In all cases, this will be the number of solutions to consider. Then multiply run-time the by time to consider each, which can be significant as well.

How to know if $O(x)$ is OK?

- As Computer Scientists, we are tempted to always look for linear or logarithmic answers. If those exist, they are of course the best!
- As contest coders, we just need to solve quickly. Check these factors:
 - How many operations can the contest computer do per second. Note this varies by language. Usually C>Java>Python
 - What are the max input sizes and max values of integers?
 - What is $O(x)$ of your algorithm?
 - What are the costs for set-up, such as initializing memory etc
- If an $O(n!)$ algorithm still works for the problem size, go for it!

Before we abandon brute force...

- Try to back-track:
 - As soon as we can, determine the solution we're constructing doesn't work and truncate the rest of the computation to consider it.
- Example, the peaceful chessboard:
How can we place N queens on a chess board without any being able to mutually attack?



Pruning the Search for Queens

- Track the constraints using simple array data structures that map to the board
- Note, the rows are kept safe explicitly with y variable

```
void search(int y) {  
    if (y == n) {  
        count++;  
        return;  
    }  
    for (int x = 0; x < n; x++) {  
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;  
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;  
        search(y+1);  
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;  
    }  
}
```

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

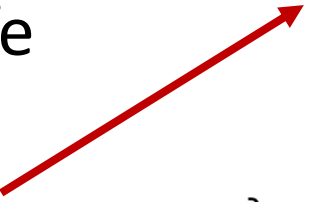
3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

Pruning the Search for Queens

- Track the constraints using simple array data structures that map to the board
- Note, the rows are kept safe explicitly with y variable
- Whenever a conflict occurs, terminate recursion

```
void search(int y) {  
    if (y == n) {  
        count++;  
        return;  
    }  
    for (int x = 0; x < n; x++) {  
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;  
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;  
        search(y+1);  
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;  
    }  
}
```



0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

Pruning the Search for Queens

- This is often fast enough for the typical 8x8 board, but for 16x16 can take up to a minute
- For queens, this is actually state-of-the-art and an open challenge.
 - Record is $q(27) = 234907967154122528$ computed only in 2016!
- For general brute force problems, we can look for even more ways to prune brute searches that might scale.

```
void search(int y) {  
    if (y == n) {  
        count++;  
        return;  
    }  
    for (int x = 0; x < n; x++) {  
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;  
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;  
        search(y+1);  
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;  
    }  
}
```

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

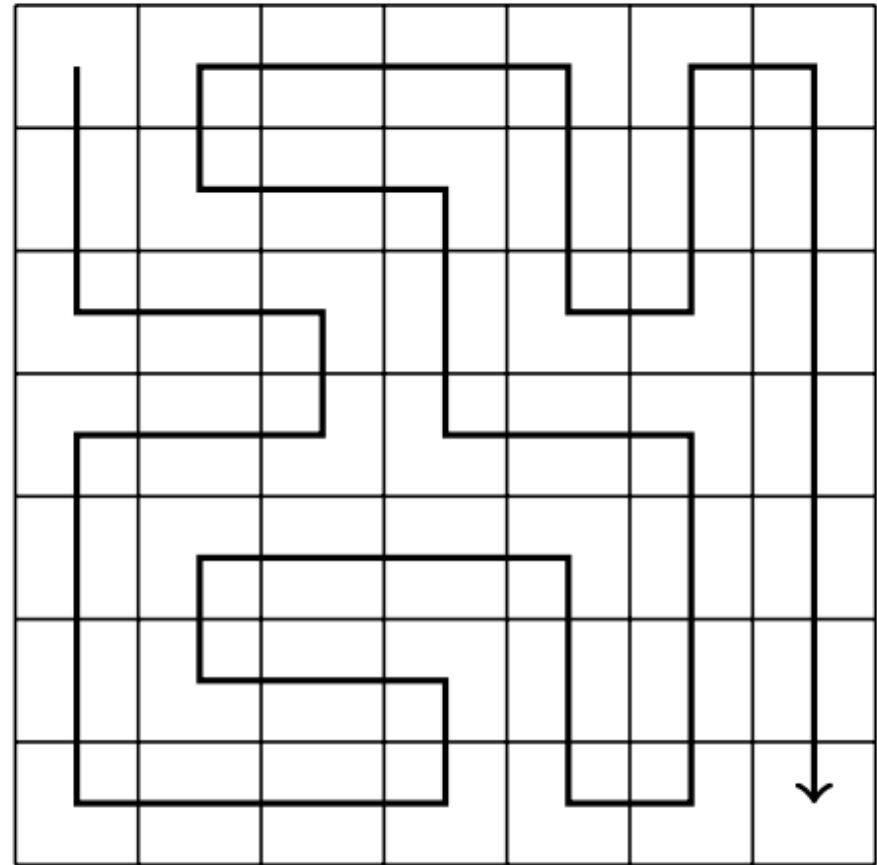
diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

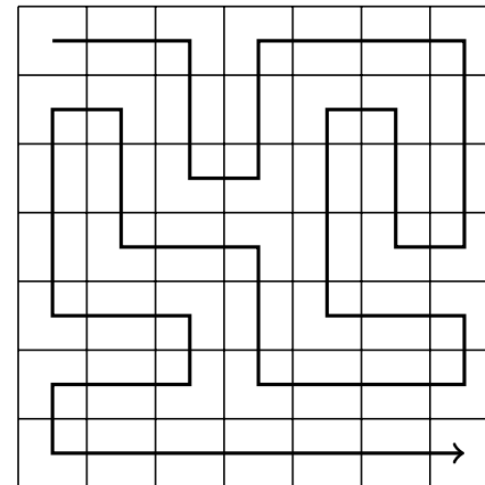
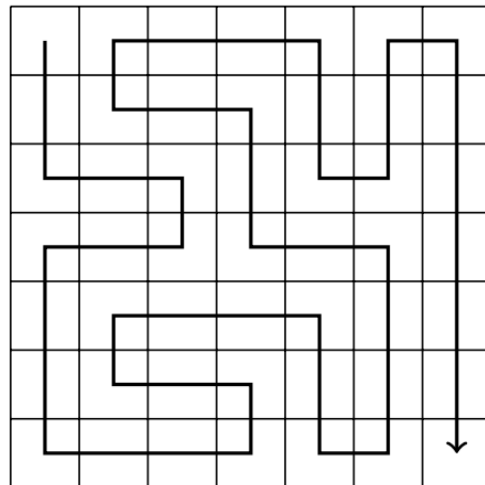
Path Counting

- Consider all paths starting in the upper left and ending in the lower right, crossing all squares exactly once.
 - The example on the right is a correct solution



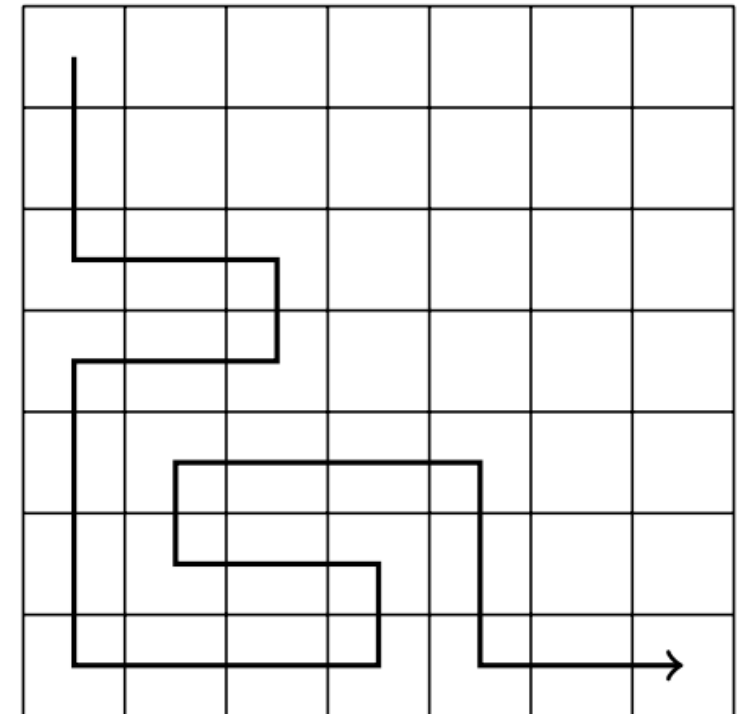
Path Counting

- Naïve solution #2: Respect symmetries in search
 - Notice that every time we move down, there is another path mirrored over the diagonal that moves right, and vice versa.
 - Only compute one or the other, then multiply the result by 2.
 - Runs in 244 seconds, 38 billion calls



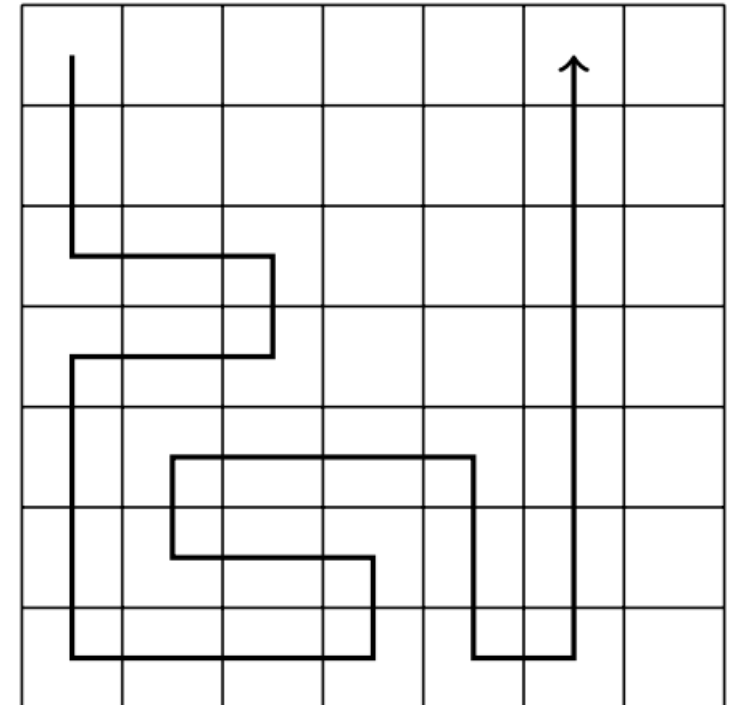
Path Counting

- First pruned solution: Special end square
 - The path has to only hit the lower right at the end, and can't come back if it gets there too early. Prune paths that open the lower right prior to the n th move.
 - Runs in 119 seconds, 20 billion calls



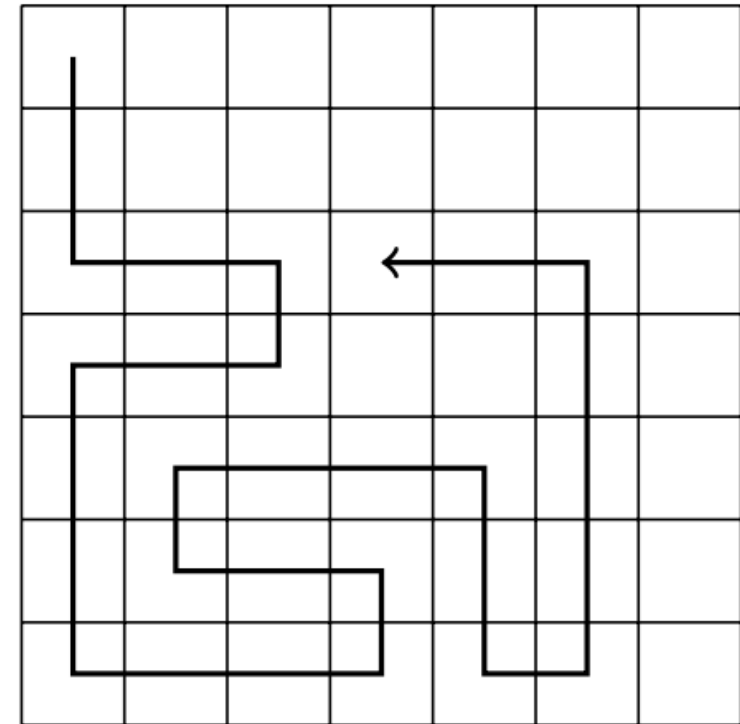
Path Counting

- Second pruned solution: Self-blocking
 - When we reach a non-corner wall with both options still open (left-right) or (up-down), then we are splitting the space in two, preventing ourselves from ever covering both options.
 - Runs in 1.8 seconds, 221 million calls



Path Counting

- Final solution: Generalize self-blocking
 - The self-blocking can also happen when we collide with our own path, and have both "turn" options open. Again the space will be split and we can never cover both sides.
 - Runs in 0.6 seconds, 69 million calls
- Final analysis: 483 -> 0.6 seconds or around 1000x speed-up from pruning! The large size of a brute force tree allows large potential for improvement
 - Good to try in contests. The code structure is still simple, but now you must be very careful about pruning correctly. (pruning away correct solutions = WA)



Greedy Algorithms

- Find an intuitive rule that guarantees a simple structure in the solution. Greedy often goes with “shortest”, “smallest”, “best” problems.
- Canonical example: Making change in well-structured currencies:
 - E.g., euro-coins have the values {1, 2, 5, 10, 20, 50, 100, 200}
 - $N=520$, optimal change is $200+200+100+20$
 - We should always use the coins in decreasing order
 - The greedy trick: prove this and ensure no edge-cases or exceptions!



Proof for Euro Coins

- Claim: each coin of $\{1, 5, 10, 50, 100\}$ only appears once:
 - By counter example: assume we had 2 of any of the above in our solution
 - Then, there exists another solution with only 1 of the next value up which is strictly better
- Claim #2: each coin of $\{2, 20\}$ appears at most twice:
 - Similarly: $2+2+2 = 5+1$, an improvement, so we would never reach 3



How to code greedy?

- Usually very simple once you know the “way to be greedy”
- Common structure:
 - Compute how “greedy” each input is, and sort them by this value
 - Start with an empty solution
 - Loop through in sorted order, either:
 - Discarding if some constraint not met (e.g., the coin value makes the change too large)
 - Else using the item in the solution
 - Return the solution in the end
- Linear complexity. Almost never TLE. Can easily be WA if your ordering doesn't guarantee optimal solution.



What about other coin selections?

- Consider the currency made up of $\{1,3,4\}$. Does our greedy algorithm work for all change amounts?



What about other coin selections?

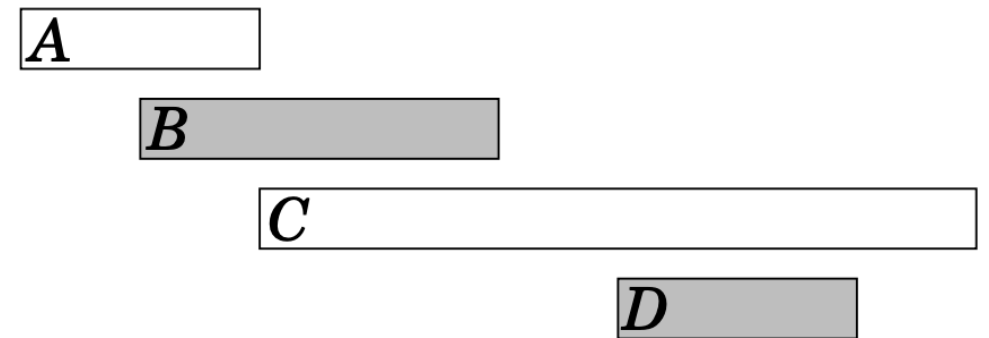
- Consider the currency made up of $\{1,3,4\}$. Does our greedy algorithm work for all change amounts?
- No, this does not preserve the optimality of the greedy approach
 - E.g., for $n=6$, the greedy solution is $4+1+1$, but clearly $3+3$ would be better.
 - Here we have no more efficient way to replace the duplicate 3's, so we cannot prove our greedy rules work.
- If the coin values are given, you can try to intuitively check the greedy pattern. For general coin problems, don't assume greedy works.
 - There is an efficient DP solution that always works: we'll see this in coming weeks.



Another Greedy Rule: find the best schedule

- Schedule as many non-overlapping events as you can
- What greedy rule works here?
 - There are a few ideas you can produce, but not all work
 - Check for counter-examples

event	starting time	ending time
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8



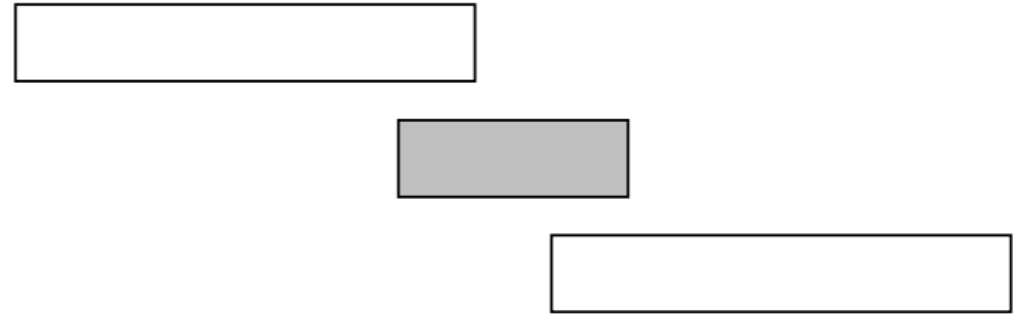
Scheduling Idea #1: Shortest

- By fitting in short events, we block less time. This seems like a good way to end up with many events?



Scheduling Idea #1: Shortest

- By fitting in short events, we block less time. This seems like a good way to end up with many events?
- This doesn't work in the example shown.



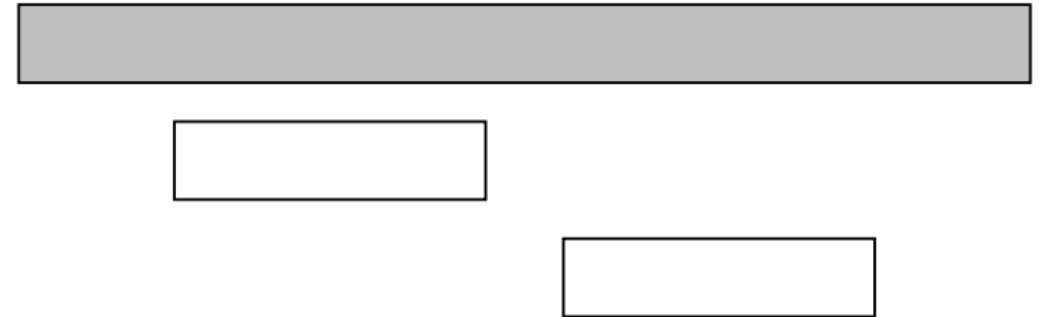
Scheduling Idea #2: Begins first

- How about being proactive. Just start with an event that's ready!



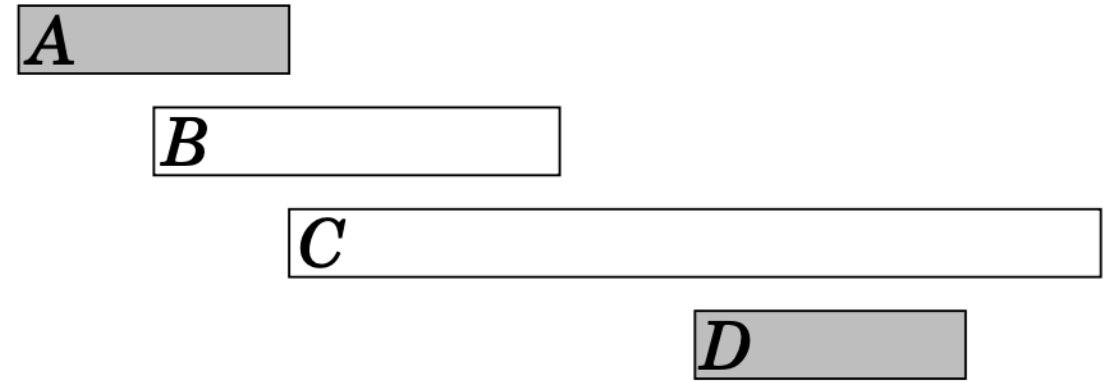
Scheduling Idea #2: Begins first

- How about being proactive. Just start with an event that's ready!
- Similarly, there is an easy counter example to find. That event ends up being boring...



Scheduling Idea #3: Ends first

- Pick the next event based on its end time.
- This one works, and you can justify it to yourself.
 - If we wouldn't select that event, by the given end-time, we have done less "work", and will just have events left with later end-times.



Tasks with deadlines

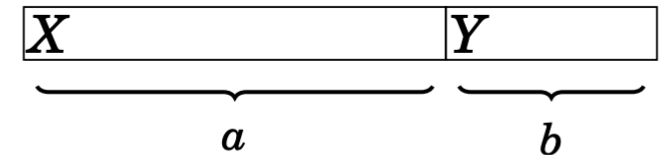
- Think CPU scheduling, but our user will be annoyed if their job isn't done by a fixed moment
- Must do all tasks, but in what order?
- Our rewards are $d-x$, how ahead/behind of the deadline are we on completion
 - Note: negative points for being late
- Claim: there is a greedy solution to this problem

task	duration	deadline
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

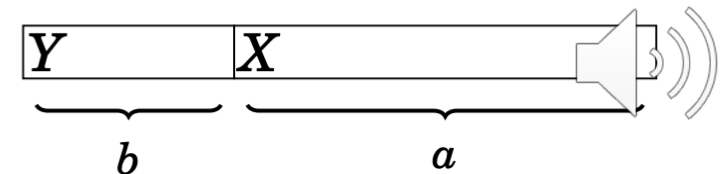


Tasks with deadlines

- Answer: sort by duration
- Optimal due to the simple “linear sum” nature of our scores
 - Although we pay a penalty for being later than the deadline, we balance with a greater reward for being earlier on all future tasks
- This of course doesn't hold for hard deadlines, cases where we can leave some tasks out, or multiple processors
- Proof of greed:
 - Consider task $a > b$ happening first
 - Swap a and b , everything gets better



Here $a > b$, so we should swap the tasks:



Greedy Algorithms are very important for real-world problems

- Beyond contests, many solutions in the computing industry involve greedy ideas.
- One example is data compression:
 - Construct a variable length codeword for each character such that we can represent strings in as few bits as possible
- Given the codebook on the right:
 - String AABACDACA
 - Encodes to 001100101110100
- How do we form the best codebook?

character	codeword
A	0
B	110
C	10
D	111



Tips

- Using Greedy solutions in programming contests is usually risky.
 - A greedy solution normally will not encounter TLE response, as it is lightweight, but tends to get WA response.
- Proving that a certain problem has optimal sub-structure and greedy property in contest time may be time consuming, so a competitive programmer usually do this.
 - Look at the input size. If it is 'small enough' for the time complexity of either Complete Search or Dynamic Programming, she will use one of these approaches as both will ensure correct answer.
 - Use Greedy solution if you know for sure that the input size is too large.
- A problem that seems extremely complicated on the surface might signal a greedy approach.



Greed on Interview Questions

- Given an input of N integers, how can we quickly arrange them in “ascending-descending” order?
 - For every odd index, both neighbors are greater
 - For every even index, both neighbors are less (or equal)
- {5, 9, 13, 2, 1, 7, 8}
 - Could become {1, 13, 2, 9, 5, 8, 7}
- Code the algorithm and discuss its run-time complexity.



Ascending-Descending

- We can clearly use sorting algorithms:
 - Sort all inputs in an efficient way
 - Produce output by alternating:
 - 1 from the front
 - 1 from the back
- This has complexity $O(n \lg(n))$. Can you do better?
 - If the interviewer asks this, think carefully about two options:
 - You already know a proof that it cannot get faster (e.g., sorting itself)
 - Is there a way to be more greedy? Often simple / think-outside-the-box



Ascending-Descending

- Linear algorithm possible by exploiting simple structure
- For each element:
 - If it mis-matches with its right neighbor, swap them
- Does this work? Think about how to prove or disprove it.



A motivating question

Abridged problem statement: Given different models for each garment (e.g. 3 shirts, 2 belts, 4 shoes, ...), *buy one model of each garment*. As the budget is *limited*, we cannot spend more money than the budget, but we want to spend *the maximum possible*. But it is also possible that we cannot buy one model of each garment due to that small amount of budget.

The input consist of two integers $1 \leq M \leq 200$ and $1 \leq C \leq 20$, where M is the budget and C is the number of garments that you have to buy. Then, there are information of the C garments. For a garment_id $\in [0 \dots C-1]$, we know an integer $1 \leq K \leq 20$ which indicates the number of different models for that garment_id, followed by K integers indicating the price of each model $\in [1 \dots K]$ of that garment_id.

The output should consist of one integer that indicates the maximum amount of money necessary to buy one element of each garment *without exceeding the initial amount of money*. If there is no solution, print “no solution”.



Public Tests

For example, if the input is like this (test case A):

$M = 20, C = 3$

3 models of garment_id 0 \rightarrow 6 4 8 // see that the prices are not sorted in input

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

Then the answer is 19, which *may* come from buying the underlined items (8+10+1).

Note that this solution is not unique, as we also have (6+10+3) and (4+10+5).

However, if the input is like this (test case B):

$M = 9$ (very limited budget), $C = 3$

3 models of garment_id 0 \rightarrow 6 4 8

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

Then the answer is “no solution” as buying all the cheapest models $(4+5+1) = 10$ is still $> M$.

Consider Solving by Brute Force

- Start with money = M and work through garment IDs in order, from 0.
- Try all models, subtracting the corresponding money and recursing to garment ID ++.
- Base case: we selected garment ID $C-1$ with money left. A candidate solution. Among these, search for the one as close to 0 money left without going over.
- In any branch, if money < 0 , prune the branch.
- What is the complexity of this algorithm?

Brute force complexity

- Each garment gives up to 20 choices, for 20 garment IDs.
- Number of selections $20 \times 20 \times \dots \times 20 = 20^{20}$ - too large for n sec limit

What about greedy?

- Since we want to maximize the budget spent, why don't we take the most expensive model in each garment_id which still fits our budget?

Does this work on our public tests?

$M = 20, C = 3$

3 models of garment_id 0 \rightarrow 6 4 8

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

$M = 9$ (very limited budget), $C = 3$

3 models of garment_id 0 \rightarrow 6 4 8

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

What about greedy?

- Since we want to maximize the budget spent, why don't we take the most expensive model in each garment_id which still fits our budget?

$M = 20, C = 3$

3 models of garment_id 0 \rightarrow 6 4 8

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

$M = 9$ (very limited budget), $C = 3$

3 models of garment_id 0 \rightarrow 6 4 8

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

Yes. So, is this the solution to code?

What about greedy?

- Since we want to maximize the budget spent, why don't we take the most expensive model in each garment_id which still fits our budget?

$$M = 12, C = 3$$

3 models of garment_id 0 \rightarrow 6 4 8

2 models of garment_id 1 \rightarrow 5 10

4 models of garment_id 2 \rightarrow 1 5 3 5

Nope. Here greedy will say “nosolution”. The optimal solution is (4+5+3 = 12), which use all our budget.

Here we need dynamic programming

- Divide and conquer, not on the greedy structure, but on common sub-problems that can be re-used:
 - Any time we have a particular value of money left, V , and are considering garment i , $S(V,i)$ is the answer from there on
 - We have at most 201 values of money left (our max budget)
 - We have 20 garment classes. Just 4020 unique problems.

Here we need dynamic programming

- Create a DP table. Use this to augment our recursion:
 - One table entry per sub-problem.
 - Each value $S(V,i)$ holds -1 initially – has not been computed
 - Before recursing, check if we already have a valid $S(V,i)$
 - If so, return it
 - If not, compute as per brute force, then update the $S(V,i)$ value to the result.
- In the next set of slides, we will look at many practical aspects of applying DP in various ways to many problems.