Combinatorial/Discrete Optimization

Abhiram Ranade

January 27, 2016

Combinatorial/Discrete optimization

Combinatorial/Discrete optimization

Subject of the rest of the course!

Combinatorial/Discrete optimization

Subject of the rest of the course!

Solution space: (Large) finite set of candidate objects.

Combinatorial/Discrete optimization

Subject of the rest of the course!

Solution space: (Large) finite set of candidate objects.

Existence problem: "Determine if there exists a candidate object satisfying a certain set of feasibility conditions."

Combinatorial/Discrete optimization

Subject of the rest of the course!

Solution space: (Large) finite set of candidate objects.

Existence problem: "Determine if there exists a candidate object satisfying a certain set of feasibility conditions."

Counting problem: "Determine the number of candidate objects satisfying feasibility conditions."

Combinatorial/Discrete optimization

Subject of the rest of the course!

Solution space: (Large) finite set of candidate objects.

Existence problem: "Determine if there exists a candidate object satisfying a certain set of feasibility conditions."

Counting problem: "Determine the number of candidate objects satisfying feasibility conditions."

Search problem: "Find a candidate object satisfying feasibility conditions."

Combinatorial/Discrete optimization

Subject of the rest of the course!

Solution space: (Large) finite set of candidate objects.

Existence problem: "Determine if there exists a candidate object satisfying a certain set of feasibility conditions."

Counting problem: "Determine the number of candidate objects satisfying feasibility conditions."

Search problem: "Find a candidate object satisfying feasibility conditions."

Optimization problem: "Find a candidate object which satisfies feasibility conditions and maximizes a certain objective function."

Example 1: n Queens problem:

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens
Specified implicitly, not by enumeration.

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens
Specified implicitly, not by enumeration.
Input: n.

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens
Specified implicitly, not by enumeration.
Input: n.

Feasibility condition: non capture

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens
Specified implicitly, not by enumeration.
Input: n.

Feasibility condition: non capture

This is a search problem.

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens
Specified implicitly, not by enumeration.
Input: n.

Feasibility condition: non capture

This is a search problem.

We could also ask for existence or count.

Example 1: n Queens problem:

Place n queens on an $n \times n$ chessboard such that no queen captures another.

Queen at (i, j) captures queen at (i', j') if one of the following holds

- $i = i'$ (same row)
- $j = j'$ (same column)
- $|i - i'| = |j - j'|$ (same "diagonal")

Solution space: all possible ways of placing n queens
Specified implicitly, not by enumeration.
Input: n.

Feasibility condition: non capture

This is a search problem.

We could also ask for existence or count.

No natural optimization problem in this case.

Example 2: Knapsack

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

- $\sum_{i \in S} w_i \leq C$ Weight capacity limit honoured

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

- $\sum_{i \in S} w_i \leq C$ — Weight capacity limit honoured
- $\sum_{i \in S} v_i$ is maximized — Maximize value

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

- $\sum_{i \in S} w_i \leq C$ — Weight capacity limit honoured
- $\sum_{i \in S} v_i$ is maximized — Maximize value

Solution space/Candidate set:

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

- $\sum_{i \in S} w_i \leq C$ — Weight capacity limit honoured
- $\sum_{i \in S} v_i$ is maximized — Maximize value

Solution space/Candidate set: All possible subsets of $\{1, \ldots, n\}$.

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

- $\sum_{i \in S} w_i \leq C$ — Weight capacity limit honoured
- $\sum_{i \in S} v_i$ is maximized — Maximize value

Solution space/Candidate set: All possible subsets of $\{1, \ldots, n\}$.

Objective function: $V(S) = \sum_{i \in S} v_i$

Example 2: Knapsack

"Pack a knapsack with most valuable items so that a total weight limit is not exceeded."

Input: Integer C, Integers $w_1, \ldots, w_n, \quad v_1, \ldots, v_n$
Output: Subset S of $\{1, \ldots, n\}$ s.t.

- $\sum_{i \in S} w_i \leq C$ — Weight capacity limit honoured
- $\sum_{i \in S} v_i$ is maximized — Maximize value

Solution space/Candidate set: All possible subsets of $\{1, \ldots, n\}$.

Objective function: $V(S) = \sum_{i \in S} v_i$

Related existence question: Does there exist a set with total value at least V, where V is an additional input to the problem.

Remarks

Remarks

- Solution space/candidate set is usually very large, typically exponential in the problem size.

Remarks

- Solution space/candidate set is usually very large, typically exponential in the problem size.
- It is possible to answer existence, search, count, optimization by systematically generating each candidate, but typically this takes exponential time.

Remarks

- Solution space/candidate set is usually very large, typically exponential in the problem size.
- It is possible to answer existence, search, count, optimization by systematically generating each candidate, but typically this takes exponential time.
- Key question: Can we do it in polynomial time?

Remarks

- Solution space/candidate set is usually very large, typically exponential in the problem size.
- It is possible to answer existence, search, count, optimization by systematically generating each candidate, but typically this takes exponential time.
- Key question: Can we do it in polynomial time?
- Feasible solution/Feasible candidate: Candidate satisfying all conditions

Remarks

- Solution space/candidate set is usually very large, typically exponential in the problem size.
- It is possible to answer existence, search, count, optimization by systematically generating each candidate, but typically this takes exponential time.
- Key question: Can we do it in polynomial time?
- Feasible solution/Feasible candidate: Candidate satisfying all conditions
- Optimal solution/Optimal candidate: Feasible solution which also maximizes objective function over all feasible solutions.

Remarks

- Solution space/candidate set is usually very large, typically exponential in the problem size.
- It is possible to answer existence, search, count, optimization by systematically generating each candidate, but typically this takes exponential time.
- Key question: Can we do it in polynomial time?
- Feasible solution/Feasible candidate: Candidate satisfying all conditions
- Optimal solution/Optimal candidate: Feasible solution which also maximizes objective function over all feasible solutions.

Instead of maximizing the objective function, it may be more natural to minimize. This is also allowed.

Exponential time solution

Exponential time solution

"Generate and test":

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive
GenerateNTest(partialSolution){

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive
GenerateNTest(partialSolution){

1. If partialSolution is complete,

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive
GenerateNTest(partialSolution){

1. If partialSolution is complete,
2.   If it satisfies feasibility report it and stop, else return

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive
GenerateNTest(partialSolution){

1. If partialSolution is complete,
2. If it satisfies feasibility report it and stop, else return
3. Else

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive
GenerateNTest(partialSolution){

1. If partialSolution is complete,
2.   If it satisfies feasibility report it and stop, else return
3. Else
4.   For each way of extending the solution

Exponential time solution

"Generate and test": Construct every possible candidate and check if it satisfies feasibility conditions.

How to generate the candidates:
Usually each candidate consists of some n parts.
Say ith part can be chosen in some q_i ways.
Total number of ways in which a candidate can be chosen:
$\prod_i q_i = \Omega(q^n)$ where $q = \min_i q_i$.

Algorithm for generating candidates: Recursive
GenerateNTest(partialSolution){

1. If partialSolution is complete,
2.   If it satisfies feasibility report it and stop, else return
3. Else
4.   For each way of extending the solution
5.   GenerateNTest(ExtendedSolution)

}

n queens

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.

n queens

Solution: Q[1..n]. Q[i] $=$ column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

n queens

Solution: Q[1..n]. Q[i] $=$ column number of row i queen.

Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3. Q[i+1] = j.

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3. Q[i+1] = j.
4. GenerateNTest(Q[1..n],i+1)

}

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3. Q[i+1] = j.
4. GenerateNTest(Q[1..n],i+1)

}

CheckFeasibility(Q[1..n]){

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3. Q[i+1] = j.
4. GenerateNTest(Q[1..n],i+1)

}

CheckFeasibility(Q[1..n]){

1. For j=2 to n

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

```
GenerateNTest(Q[1..n],i){
1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3.         Q[i+1] = j.
4.         GenerateNTest(Q[1..n],i+1)
}
```

```
CheckFeasibility(Q[1..n]){
1. For j=2 to n
2.     For i=1 to j-1
```

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3. Q[i+1] = j.
4. GenerateNTest(Q[1..n],i+1)

}

CheckFeasibility(Q[1..n]){

1. For j=2 to n
2. For i=1 to j-1
3. if Q[i]=Q[j] or $|Q[i] - Q[j]|$ = j-i return false

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

GenerateNTest(Q[1..n],i){

1. If i = n, then check feasibility and stop if feasible.
2. Else For j=1 to n
3. Q[i+1] = j.
4. GenerateNTest(Q[1..n],i+1)

}

CheckFeasibility(Q[1..n]){

1. For j=2 to n
2. For i=1 to j-1
3. if Q[i]=Q[j] or $|Q[i] - Q[j]|$ = j-i return false
4. return true

}

n queens

Solution: Q[1..n]. Q[i] = column number of row i queen.
Partial solution: Q[1..n], i. Queens placed in first i rows.

```
GenerateNTest(Q[1..n],i){
  1. If i = n, then check feasibility and stop if feasible.
  2. Else For j=1 to n
  3.         Q[i+1] = j.
  4.         GenerateNTest(Q[1..n],i+1)
}
```

```
CheckFeasibility(Q[1..n]){
  1. For j=2 to n
  2.     For i=1 to j-1
  3.     if Q[i]=Q[j] or |Q[i] − Q[j]| = j-i return false
  4. return true
}
```

"Lexicographic exploration". Other orders possible.

Knapsack

Knapsack

Solution: x[1..n] : x[i] $=$ 0 or 1 indicating whether ith object put in knapsack.

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Feasibility: Weight must be less than capacity

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Feasibility: Weight must be less than capacity

Objective function: Total value

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Feasibility: Weight must be less than capacity

Objective function: Total value

Modification to algorithm:

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Feasibility: Weight must be less than capacity

Objective function: Total value

Modification to algorithm:
Check feasibility. If feasible, evaluate objective. If objective value of current solution is more than the previous best, replace previous best.

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Feasibility: Weight must be less than capacity

Objective function: Total value

Modification to algorithm:
Check feasibility. If feasible, evaluate objective. If objective value of current solution is more than the previous best, replace previous best.

"Best solution": Store in a global variable.

Knapsack

Solution: x[1..n] : x[i] = 0 or 1 indicating whether ith object put in knapsack.

x[i] : "indicator variables"

Partial solution: x[1..i]

Feasibility: Weight must be less than capacity

Objective function: Total value

Modification to algorithm:
Check feasibility. If feasible, evaluate objective. If objective value of current solution is more than the previous best, replace previous best.

"Best solution": Store in a global variable.

Next Homework: Write a program based on the above ideas.

Remarks

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Will it help to generate solutions in other orders besides lexicographic?

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Will it help to generate solutions in other orders besides lexicographic?

- Example: In n queens, start placing queens in row order n/2,n/2-1,n/2+1,...

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Will it help to generate solutions in other orders besides lexicographic?

- Example: In n queens, start placing queens in row order n/2,n/2-1,n/2+1,...
- Example: Place the ith row queen in odd numbered columns first and then even numbered.

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Will it help to generate solutions in other orders besides lexicographic?

- Example: In n queens, start placing queens in row order n/2,n/2-1,n/2+1,...
- Example: Place the ith row queen in odd numbered columns first and then even numbered.

Heuristics such as these often speed up.

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Will it help to generate solutions in other orders besides lexicographic?

- Example: In n queens, start placing queens in row order n/2,n/2-1,n/2+1,...
- Example: Place the ith row queen in odd numbered columns first and then even numbered.

Heuristics such as these often speed up.

Sometimes we can even prove that everything finishes in polytime!

Remarks

- Generation usually takes exponential time because there are exponential number of candidates.
- Feasibility testing usually is simple and takes polytime.
- Overall time : exponential.
- Time to find a single solution can vary quite a bit. It could be the first complete candidate generated, or the last.

Will it help to generate solutions in other orders besides lexicographic?

- Example: In n queens, start placing queens in row order n/2,n/2-1,n/2+1,...
- Example: Place the ith row queen in odd numbered columns first and then even numbered.

Heuristics such as these often speed up.

Sometimes we can even prove that everything finishes in polytime!

Next few weeks..

Speeding up knapsack

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i.

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

If space remains, pick item having next largest v_i/w_i.

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i / w_i

If space remains, pick item having next largest v_i / w_i.

...

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

If space remains, pick item having next largest v_i/w_i.

...

Conjecture: We get the optimal solution.

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

If space remains, pick item having next largest v_i/w_i.

...

Conjecture: We get the optimal solution.
Unfortunately this is false. Can you give a counter example?

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

If space remains, pick item having next largest v_i/w_i.

...

Conjecture: We get the optimal solution.
Unfortunately this is false. Can you give a counter example?

C=10, w={10,6}, v={10,7}

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

If space remains, pick item having next largest v_i/w_i.

...

Conjecture: We get the optimal solution.
Unfortunately this is false. Can you give a counter example?

C=10, w={10,6}, v={10,7}
Greedy solution: item 2. Value 7.

Speeding up knapsack

Could we prove that items possessing certain properties will always be picked in the optimal knapsack?

"Greedy" Heuristic: If we pick item i we use up capacity w_i but gain value v_i. Why not try to maximize our gain per capacity use?

First pick the item having max v_i/w_i

If space remains, pick item having next largest v_i/w_i.

...

Conjecture: We get the optimal solution.
Unfortunately this is false. Can you give a counter example?

C=10, w={10,6}, v={10,7}
Greedy solution: item 2. Value 7.
Optimal solution: item 1. Value 10.

Fractional Knapsack

Fractional Knapsack

Input: Capacity, weights, values as before.

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}
Solution is item 2 fully, and 4/10 fraction of item 1. Value: 11.

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}
Solution is item 2 fully, and 4/10 fraction of item 1. Value: 11.

Theorem: Greedy strategy gives optimal solution for fractional knapsack.

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}
Solution is item 2 fully, and 4/10 fraction of item 1. Value: 11.

Theorem: Greedy strategy gives optimal solution for fractional knapsack.

Proof idea:

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}
Solution is item 2 fully, and 4/10 fraction of item 1. Value: 11.

Theorem: Greedy strategy gives optimal solution for fractional knapsack.

Proof idea:
Suppose I make a "small change" to the current solution.

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}
Solution is item 2 fully, and 4/10 fraction of item 1. Value: 11.

Theorem: Greedy strategy gives optimal solution for fractional knapsack.

Proof idea:
Suppose I make a "small change" to the current solution.
The value should not increase.

Fractional Knapsack

Input: Capacity, weights, values as before.
Output: Pick objects to maximize values. Objects can be picked fractionally.

What happens for instance: C=10, w={10,6}, v={10,7}
Solution is item 2 fully, and 4/10 fraction of item 1. Value: 11.

Theorem: Greedy strategy gives optimal solution for fractional knapsack.

Proof idea:
Suppose I make a "small change" to the current solution.
The value should not increase.
Use this to deduce the form of an optimal solution.

Proof

Proof

Suppose item i is included to fraction f_i in optimal solution.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$

New fractions give feasible solution.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$

New fractions give feasible solution.
Change in value: $-v_k \epsilon + v_j \epsilon w_k / w_j$

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$
New fractions give feasible solution.
Change in value: $-v_k \epsilon + v_j \epsilon w_k / w_j \leq 0$ $\because$ Original was optimal

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$
New fractions give feasible solution.
Change in value: $-v_k \epsilon + v_j \epsilon w_k / w_j \leq 0$ $\because$ Original was optimal
Thus $v_j / w_j \leq v_k / w_k$.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$
New fractions give feasible solution.
Change in value: $-v_k \epsilon + v_j \epsilon w_k / w_j \leq 0$ $\because$ Original was optimal
Thus $v_j / w_j \leq v_k / w_k$.
Contrapositive: If $v_j / w_j > v_k / w_k$ in an optimal solution, then either $f_j = 1$ or $f_k = 0$.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$
New fractions give feasible solution.
Change in value: $-v_k \epsilon + v_j \epsilon w_k / w_j \leq 0$ $\because$ Original was optimal
Thus $v_j / w_j \leq v_k / w_k$.
Contrapositive: If $v_j / w_j > v_k / w_k$ in an optimal solution, then either $f_j = 1$ or $f_k = 0$.

Items with higher v/w must be fully selected before items with lower v/w are considered.

Proof

Suppose item i is included to fraction f_i in optimal solution.

Suppose in optimal $f_j < 1, f_k > 0$ for some $j \neq k$.
Small change: replace a bit of item k by a bit of item j.
We throw out ϵ fraction of item k. Need $f_k - \epsilon \geq 0$
This frees ϵw_k capacity.
Add fraction $\epsilon w_k / w_j$ of item j. Need $f_j + \epsilon w_k / w_j \leq 1$
Choose $\epsilon = \min(f_k, (1 - f_j) w_j / w_k)$
New fractions give feasible solution.
Change in value: $-v_k \epsilon + v_j \epsilon w_k / w_j \leq 0$ $\because$ Original was optimal
Thus $v_j / w_j \leq v_k / w_k$.
Contrapositive: If $v_j / w_j > v_k / w_k$ in an optimal solution, then either $f_j = 1$ or $f_k = 0$.

Items with higher v/w must be fully selected before items with lower v/w are considered.

Choose items in non-increasing order of v/w.

Summary

Summary

- Finding a solution $=$ making a sequence of decisions.

Summary

- Finding a solution = making a sequence of decisions.
- Every decision can be thought of as having some "short term benefit" and "short term cost"

Summary

- Finding a solution = making a sequence of decisions.
- Every decision can be thought of as having some "short term benefit" and "short term cost"
- "Short term" : current choice may prevent some other choices later. The loss because of that restriction is the long term loss.

Summary

- Finding a solution = making a sequence of decisions.
- Every decision can be thought of as having some "short term benefit" and "short term cost"
- "Short term" : current choice may prevent some other choices later. The loss because of that restriction is the long term loss.
- Greedy strategy: maximize short term benefit/short term cost

Summary

- Finding a solution = making a sequence of decisions.
- Every decision can be thought of as having some "short term benefit" and "short term cost"
- "Short term" : current choice may prevent some other choices later. The loss because of that restriction is the long term loss.
- Greedy strategy: maximize short term benefit/short term cost
- Greedy strategy may not always work. So need a proof.

Summary

- Finding a solution = making a sequence of decisions.
- Every decision can be thought of as having some "short term benefit" and "short term cost"
- "Short term" : current choice may prevent some other choices later. The loss because of that restriction is the long term loss.
- Greedy strategy: maximize short term benefit/short term cost
- Greedy strategy may not always work. So need a proof.
- Proof can use the "exchange argument".

Summary

- Finding a solution = making a sequence of decisions.
- Every decision can be thought of as having some "short term benefit" and "short term cost"
- "Short term" : current choice may prevent some other choices later. The loss because of that restriction is the long term loss.
- Greedy strategy: maximize short term benefit/short term cost
- Greedy strategy may not always work. So need a proof.
- Proof can use the "exchange argument".
- Exchange argument might also help us discover the greedy strategy.

Remark

Remark

Our strategy was: pick items in decreasing order of v/w.

Remark

Our strategy was: pick items in decreasing order of v/w.

Can this be specified recursively?

Remark

Our strategy was: pick items in decreasing order of v/w.

Can this be specified recursively?

Yes: Pick the item with the largest v/w. Then recurse.

Remark

Our strategy was: pick items in decreasing order of v/w.

Can this be specified recursively?

Yes: Pick the item with the largest v/w. Then recurse.

The recursive version explicitly talks only about picking one item. This might be often easier to reason with – the usual benefit of recursion.