NP complete problems

Some figures, text, and pseudocode from:
- Introduction to Algorithms, by Cormen, Leiserson, Rivest and Stein
- Algorithms, by Dasgupta, Papadimitriou, and Vazirani
Module objectives

• Some problems are too hard to solve in polynomial time
  – Example of such problems, and what makes them hard

• Class NP\P
  – NP: problems with solutions verifiable in poly time
  – P: problems not solvable in poly time

• NP-complete, fundamental class in Computer Science
  – reduction form on problem to another

• Approximation Algorithms:
  – since these problems are too hard, will settle for non-optimal solution
  – but close to the optimal
  – if we can find such solution reasonably fast
Module objectives

• **WARNING:** This presentation trades rigor for intuition and easiness

• The CLRS book ch 35 is rigorous, but considerably harder to read
  
  – hopefully easier after going through these slides

• For an introduction to complexity theory that is rigorous and somewhat more accessible, see
  
  – Michael Sipser: *Introduction to Theory of Computation*
2SAT problem

- **2-clause** \((a \lor b)\)
  - true (satisfied) if either \(a\) or \(b\) true, false (unsatisfied) if both false
  - \(a, b\) are binary true/false literals
  - \(a = \text{not} (a) = \text{negation} (a). \quad \neg T=F \ ; \quad \neg F=T\)
  - can have several clauses, e.g. \((a \lor b), (\neg a \lor c), (\neg c \lor d), (\neg a \lor \neg b)\)
  - truth table for logical OR: \((T \lor T)=T; (T \lor F)=T; (F \lor T)=T; (F \lor F)=F\)

- **2-SAT problem**: given a set of clauses, find an assignment \(T/F\) for literals in order to satisfy all clauses
Example: satisfy the following clauses:
- \((a \lor b) \land (\neg a \lor c) \land (\neg d \lor b) \land (d \lor \neg c) \land (\neg c \lor f) \land (\neg f \lor \neg g) \land (g \lor \neg d)\)

**try a=TRUE**
- \(a=T \Rightarrow \neg a=F \Rightarrow c=T \Rightarrow d=f=T \Rightarrow \neg g=T \Rightarrow g=F \Rightarrow \neg d=T\) contradiction

**try a=FALSE**
- \(a=F \Rightarrow b=T\), it works; eliminate first three clauses and \(a, b\); now we have \((d \lor \neg c) \land (\neg c \lor f) \land (\neg f \lor \neg g) \land (g \lor \neg d)\)

**try c=FALSE**
- it works, eliminate first two clauses and \(c\), remaining \((\neg f \lor \neg g) \land (g \lor \neg d)\)

**try g=TRUE**
- \(g=T \Rightarrow \neg g=F \Rightarrow \neg f=T\); done.

**assignment**: TRUE(b, g) ; FALSE(a, c, f), EITHER (d)
2SAT algorithm

- pick one literal not assigned yet, say “a”, from a clause still to be satisfied
  - see if THINGS_WORK_OUT( a ) //try assign a=TRUE
  - if NOT, see if THINGS_WORK_OUT( ¬a ) // try assign a=FALSE

- if still NOT, return “NOT POSSIBLE”

- if YES (either way), keep the assignments made, and delete all clauses that are satisfied by assignments

- repeat from the beginning until there are no clauses left, or until “NOT POSSIBLE” shows up
How to try an assignment for 2SAT

THINGS_WORK_OUT (a)

- queue Q={a}

- while x=dequeue(Q)
  - for each clause that contain ¬x like (y∨¬x) or (¬x∨y):
    - if y=FALSE (or ¬y=TRUE) already assigned, return “NOT POSSIBLE”
    - assign y=TRUE (or ¬y=FALSE), enqueue(y,Q)

- return the list of TRUE/FALSE assignments made.
2SAT algorithm

- Running time: more than linear in number of clauses, if we are unlucky
  - Easy to implement
  - \( n = \) number of literals, \( c = \) number of clauses.
  - Definitely polynomial, less than \( O(nc) \)
  - 2SAT can be solved in linear time using graph path search

- 2SAT-MAX: if an instance to 2-SAT is not satisfiable, satisfy as many clauses as possible
  - This problem is much harder, “NP-hard”
3SAT

- CLRS book calls it “3-CNF satisfiability”
- same as 2SAT, but clauses contain 3 literals
  - example \((a \lor b \lor \neg c), (\neg b \lor c \lor \neg a), (d \lor c \lor b), (\neg d \lor e \lor c), (\neg e \lor b \lor d)\)
- try to solve/satisfy this problem with an intelligent/fast algorithm – can’t find such a solution
  - exercise: why THINGS_WORK_OUT procedure is not applicable on 3SAT?
- this problem can be solved only by essentially trying [almost] all possibilities
  - even if done efficiently, still an exponential time/trials
- why is 3SAT problem so hard?
why is 3SAT hard?
- no one knows for sure, but widely believe to be true (no proof yet)
- the answer seems to be that on problems that solution come from an exponential space
- not enough space structure to search efficiently (polynomial time)

proving either
- that no polynomial solution exists for 3SAT
- or finding a polynomial solution for 3SAT

... would make you rich and very famous
class $\text{NP} = \text{polynomial verification}$

- 2SAT, 3SAT very different for finding a solution
- but 2SAT, 3SAT same for verifying a solution: if someone proposes a solution, it can be verified immediately
  - proposed solution = all literals assigned T/F
  - just check every clause to be TRUE
- $\text{NP} = \text{problems for which possible solutions can be verified quickly (polynomial)}$
- $\text{P} = \text{problems for which solutions can be found quickly}$
  - obviously $\text{P} \subseteq \text{NP}$, since finding a solution is harder than verifying one
  - 2SAT, 3SAT $\in \text{NP}$
  - 2SAT $\in \text{P}$, 3SAT $\not\in \text{P}$
problems in NP/P

- NP/P problems: solutions are quickly verifiable, but hard to find
  - like 3SAT
  - also CIRCUIT-SAT,
  - CLIQUE
  - VERTEX-COVER
  - HAMILTONIAN-CYCLE
  - TSP
  - SUBSET-SUM
  - many many others, generally problems asking "find the subset that maximizes ..."
NP-reduction

- problem A reduces to problem B if
  - any input x for pb A \(\rightarrow\) input y for pb B
  - solution/answer for (y,B) \(\rightarrow\) solution/answer for (x,A)
  - “map” has to be done in polynomial time
  - A \(\text{poly-map}\rangle\) B or A \(\leq_p\) B (\(\leq_p\) stands for “polynomial-easier-than”)

- think “B harder than A”, since solving B means also solving to A via reduction

- 3SAT reduces to CLIQUE
  - 3SAT \(\leq_p\) CLIQUE

- CLIQUE reduces to VERTEX-COVER
  - CLIQUE \(\leq_p\) VERTEX-COVER
reductions

CIRCUIT-SAT
  ↓
  SAT
  ↓
  3-CNF-SAT
  ↓
  CLIQUE
  ↓
  VERTEX-COVER
  ↓
  HAM-CYCLE
  ↓
  TSP

SUBSET-SUM
CLIQUE problem

- A clique in an undirected graph $G=(V,E)$ is a set of vertices $S \subseteq V$ in which all edges exist: $\forall u,v \in S \ (u,v) \in E$
  - A clique of size $n$ must have all $(n \ choose \ 2)$ edges

- Task: find the maximal set $S$ that is a clique
CLIQUE problem

● a clique in undirected graph \( G=(V,E) \) is a set of vertices \( S \subseteq V \) in which all edges exist: \( \forall u,v \in S \ (u,v) \in E \)

  a clique of size \( n \) must have all \( \binom{n}{2} \) edges

● Task: find the maximal set \( S \) that is a clique

● in the picture, two cliques are shown of size 3 and 4
CLIQUE problem

- a clique in undirected graph \( G=(V,E) \) is a set of vertices \( S \subseteq V \) in which all edges exist: \( \forall u,v \in S \ (u,v) \in E \)
  - a clique of size \( n \) must have all \( \binom{n}{2} \) edges

- Task: find the maximal set \( S \) that is a clique

- in the picture, two cliques are shown of size 3 and 4
- the maximal clique is of size 4, as no clique of size 5 exists
CLIQUE problem

- a clique in undirected graph $G=(V,E)$ is a set of vertices $S \subseteq V$ in which all edges exist: $\forall u,v \in S \ (u,v) \in E$

  - a clique of size $n$ must have all $(n \text{ choose } 2)$ edges

- Task: find the maximal set $S$ that is a clique

- in the picture, two cliques are shown of size 3 and 4

- the maximal clique is of size 4, as no clique of size 5 exists

- CLIQUE is hard to solve: we dont know any efficient algorithm to search for cliques.
3SAT reduces to CLIQUE

- idea: for the $K$ clauses input to 3SAT, draw literals as vertices, and all edges between vertices except
  - across clauses only (no edges inside a clause)
  - not between $x$ and $\neg x$

- reduction takes poly time
- a satisfiable assignment $\Rightarrow$ a clique of size $K$
- a clique of size $K$ $\Rightarrow$ satisfiable assignment
VERTEX COVER

- Graph undirected $G = (V,E)$
- Task: find the minimum subset of vertices $T \subset V$, such that any edge $(u,v) \in E$ has at least one end $u$ or $v$ in $T$.
- NP-hard
CLIQUE reduces to VERTEX-COVER

- idea: start with graph \( G=(V,E) \) input of the CLIQUE problem
- construct the complement graph \( G'=(V,E') \) by only considering the missing edges from \( E: E' = \{\text{all } (u,v)\} \setminus E \)

  poly time reduction

- clique of size \( K \) in \( G \) \( \Rightarrow \) vertex cover of size \( |V|-k \) in \( G' \)
- vertex cover of size \( k \) in \( G' \) \( \Rightarrow \) clique of size \( |V|-K \) in \( G \)
SUBSET-SUM problem

- Given a set of positive integers $S=\{a_1,a_2,\ldots,a_n\}$ and an integer size $t$
- Task: find a subset of numbers from $S$ that sum to $t$
  - there might be no such subset
  - there might be multiple subsets
- Close related to discrete Knapsack (module 7)
3SAT reduction to SUBSET-SUM

- poly-time reduction
- SUBSET-SUM is NP complete
- CLRS book 34.5.5

Figure 34.19  The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is \( \phi = C_1 \land C_2 \land C_3 \land C_4 \), where \( C_1 = (x_1 \lor \neg x_2 \lor \neg x_3) \), \( C_2 = (\neg x_1 \lor \neg x_2 \lor \neg x_3) \), \( C_3 = (\neg x_1 \lor \neg x_2 \lor x_3) \), and \( C_4 = (x_1 \lor x_2 \lor x_3) \). A satisfying assignment of \( \phi \) is \( \langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle \). The set \( S \) produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, \( S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\} \). The target \( t \) is 1114444. The subset \( S' \subseteq S \) is lightly shaded, and it contains \( v'_1, v'_2, \) and \( v_3 \), corresponding to the satisfying assignment. It also contains slack variables \( s_1, s'_1, s'_2, s_3, s_4, \) and \( s'_4 \) to achieve the target value of 4 in the digits labeled by \( C_1 \) through \( C_4 \).
NP complete problems

- Problem A is NP-complete if
  - A is in NP (poly-time to verify proposed solution)
  - Any problem in NP reduces to A

- Second condition says: if one solves pb A, it solves via polynomial reductions all other problems in NP

- Circuit SAT is NP-complete (see book)
  - And so the other problems discussed here, because they reduce to it

- NP-complete contains as of 2013 thousands well known “apparently hard” problems
  - Unlikely one (same as “all”) of them can be solved in poly time...
  - That would mean P=NP, which many believe not true.
P vs NP problem

- see book for co-NP class definition
- four possibilities, no one knows which one is true
- most believe (d) to be true
- prove P=NP: find a poly time solver for an NP-complete pb, for ex 3SAT
- prove P≠NP: prove that an NP-complete pb cant have poly-time solver
Approximation Algorithms
Some problems too hard

• ... to solve exactly
• so we settle for a non-optimal solution
• use an efficient algorithm, sometime Greedy
• solution wont be optimal, but how much non-optimal?
  - objective(SOL) VS objective(OPTSOL)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V\)Cover
  - delete all edges with ends in \(u\) or \(v\)

- repeat until no edges left

for the example in the picture:
**Vertex Cover approx algorithm**

- choose an edge \((u,v)\)
  - add \(u,v\) to VCover
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{Cover}\)
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to VCover
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge (u,v)
  - add u, v to VCover
  - delete all edges with ends in u or v
- repeat until no edges left
- for the example in the picture:
  - (a, i)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{\text{Cover}}\)
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V\text{Cover}\)
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to VCover
  - delete all edges with ends in \(u\) or \(v\)

- repeat until no edges left

- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V\)Cover
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to VCover
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{Cover}\)
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
**Vertex Cover approx algorithm**

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{Cover}\)
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
Vertex Cover approx algorithm

- Choose an edge \((u,v)\)
  - Add \(u,v\) to \(V\text{Cover}\)
  - Delete all edges with ends in \(u\) or \(v\)
- Repeat until no edges left
- For the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{\text{Cover}}\)
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
**Vertex Cover approx algorithm**

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{\text{Cover}}\)
  - delete all edges with ends in \(u\) or \(v\)

- repeat until no edges left

- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to VCover
  - delete all edges with ends in \(u\) or \(v\)
- repeat until no edges left
- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V_{Cover}\)
  - delete all edges with ends in \(u\) or \(v\)

- repeat until no edges left

- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
  - \((e,f)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to VCover
  - delete all edges with ends in \(u\) or \(v\)

- repeat until no edges left

- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
  - \((e,f)\)
Vertex Cover approx algorithm

- choose an edge \((u,v)\)
  - add \(u,v\) to \(V\)Cover
  - delete all edges with ends in \(u\) or \(v\)

- repeat until no edges left

- for the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
  - \((e,f)\)
**Vertex Cover approx algorithm**

- Choose an edge \((u,v)\)
  - Add \(u,v\) to \(V\)Cover
  - Delete all edges with ends in \(u\) or \(v\)
- Repeat until no edges left
- For the example in the picture:
  - \((a,i)\)
  - \((h,j)\)
  - \((b,c)\)
  - \((e,f)\)

\[\text{VC}_{\text{approx}} = \{a,i,h,j,b,c,e,f\}\]

\[\text{VC}_{\text{OPTIM}} = \{b,d,e,g,k,i,h\}\]
Vertex Cover approx algorithm

1. Choose an edge \((u,v)\)
   - Add \(u\) and \(v\) to \(V_{\text{Cover}}\)
   - Delete all edges with ends in \(u\) or \(v\)
2. Repeat until no edges left
3. For the example in the picture:
   - \((a,i)\)
   - \((h,j)\)
   - \((b,c)\)
   - \((e,f)\)
4. \(V_{\text{approx}} = \{a,i,h,j,b,c,e,f\}\)
5. \(V_{\text{opt}} = \{b,d,e,g,k,i,h\}\)

**Theorem:**

\[ \text{size}(V_{\text{approx}}) \leq \text{size}(V_{\text{opt}}) \times 2 \]

- Approx ratio of 2
Set Cover problem

- set of towns $S = \{a,b,c,d,...,k\}$
- edge$(u,v) : \text{distance}(u,v) < 10\text{miles}$
- Set Cover $SC \subseteq S :$ a set of towns such that every town is within 10 miles of some town in $SC$
Set Cover problem

- set of towns $S = \{a, b, c, d, \ldots, k\}$
- edge$(u, v) : \text{distance}(u, v) < 10\text{miles}$
- Set Cover $SC \subseteq S :$ a set of towns such that every town is within 10 miles of some town in $SC$
- $S = \{a, b, e, i\}$ is a set cover
  - every town within 10 miles of one in $S$
Set Cover problem

- set of towns $S = \{a, b, c, d, \ldots, k\}$
- edge$(u, v)$ : distance$(u, v) < 10$ miles
- Set Cover $SC \subseteq S :$ a set of towns such that every town is within 10 miles of some town in $SC$
- $S = \{a, b, e, i\}$ is a set cover
  - every town within 10 miles of one in $S$
- $S = \{i, e, c\}$ a smaller set cover
Set Cover problem

- set of towns $S = \{a,b,c,d,...,k\}$
- edge($u,v$) : distance($u,v$)<10miles
- Set Cover $SC \subset S$ : a set of towns such that every town is within 10 miles of some town in $SC$
- $S = \{a,b,e,i\}$ is a set cover
  - every town within 10miles of one in $S$
- $S= \{i,e,c\}$ a smaller set cover
- TASK: find minimum size SetCover
  - NP complete
  - general version of Vertex Cover
Set Cover approx algorithm
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - $\text{deg}(a) = 6$
  - eliminate "a" and all "a"-neighbors
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - \( \text{deg}(a) = 6 \)
  - eliminate "a" and all "a"-neighbors
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - $\text{deg}(a) = 6$
  - eliminate “a” and all “a”-neighbors

- pick the next vertex with most connections to uncovered towns
  - $\text{deg}_{\text{now}}(g) = 1$
  - eliminate g and g-neighbors
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - $\text{deg}(a)=6$
  - eliminate “a” and all “a”-neighbors

- pick the next vertex with most connections to uncovered towns
  - $\text{deg}_{\text{now}}(g)=1$
  - eliminate g and g-neighbors
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - \( \text{deg}(a) = 6 \)
  - eliminate "a" and all "a"-neighbors

- pick the next vertex with most connections to uncovered towns
  - \( \text{deg}_\text{now}(g) = 1 \)
  - eliminate g and g-neighbors

- repeat for j then for c
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - $\text{deg}(a) = 6$
  - eliminate “$a$” and all “$a$”-neighbors

- pick the next vertex with most connections to uncovered towns
  - $\text{deg}_{\text{now}}(g) = 1$
  - eliminate $g$ and $g$-neighbors

- repeat for $j$ then for $c$
Set Cover approx algorithm

- pick the vertex with most connections/degree
  - deg(a)=6
  - eliminate “a” and all “a”-neighbors
- pick the next vertex with most connections to uncovered towns
  - deg_now(g)=1
  - eliminate g and g-neighbors
- repeat for j then for c
- VertexCover = \{a, g, j, c\}, size 4
Set Cover approx algorithm

- SetCover_approx = \{a,j,c,g\}, size 4
- SetCover_optimal = \{b,i,e\}, size 3
Set Cover approx algorithm

- SetCover_approx = \{a,j,c,g\}, size 4
- SetCover_optimal = \{b,i,e\}, size 3

Theorem:

\[ \text{size}(\text{SetCover\_greedy}) \leq \text{size}(\text{SetCover\_optimal}) \times \log(|V|) \]

- approx ratio is \( \log(n) \)
CLIQUE approximation

- much harder to approximate CLIQUE than VECTOR-COVER
- see wikipedia CLIQUE page
- there can be no polynomial time algorithm that approximates the maximum clique to within a factor better than $O(n^{1 - \varepsilon})$, for any $\varepsilon > 0$
3SAT approximation algorithm

• simple algorithm: assign each literal to TRUE or FALSE randomly, independently

• success: for any 3SAT clause \((a \lor b \lor c)\) the probability of evaluating FALSE is computed as the probability of all three literals to be FALSE

  \[
p[(a \lor b \lor c) = \text{FALSE}] = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}
\]

• we can expect about \(\frac{7}{8}\) of the clauses to be satisfied and \(\frac{1}{8}\) to be not satisfied

• approx rate (expected) \(\frac{8}{7}\)
SUBSET-SUM problem

- Given a set of positive integers $S = \{a_1, a_2, \ldots, a_n\}$ and an integer size $T$
  - Task: find a subset of numbers from $S$ that sum to $t$
- Idea: while traversing the array, keep a list with all partial sums
  - index 0: $L_0 = \{0\}$
  - index 1: $L_1 = \{0, a_1\}$
  - index 2: $L_2 = \{0, a_1, a_2, a_1+a_2\}$
  - index 3: $L_3 = \{0, a_1, a_2, a_3, a_1+a_2, a_1+a_3, a_2+a_3, a_1+a_2+a_3\}$
- at index $n$, verify if $T$ is in the final list
SUBSET SUM exact algorithm

```
EXACT-SUBSET-SUM(S, t)
1  n = |S|
2  L_0 = \langle 0 \rangle
3  for i = 1 to n
4     L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5     remove from L_i every element that is greater than t
6  return the largest element in L_n
```

- exponential running time!
  - because the list L_i size can become exponential

- exercise: compare with DP solution based on discrete Knapsack
SUBSET SUM approx algorithm

APPROX-SUBSET-SUM(S, t, ε)

1  \[ n = |S| \]
2  \[ L_0 = \langle 0 \rangle \]
3  \[ \text{for } i = 1 \text{ to } n \]
4  \[ L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i) \]
5  \[ L_i = \text{TRIM}(L_i, \epsilon/2n) \]
6  \[ \text{remove from } L_i \text{ every element that is greater than } t \]
7  \[ \text{let } z^* \text{ be the largest value in } L_n \]
8  \[ \text{return } z^* \]

- \text{TRIM}(L, \epsilon/2n) truncates long lists to avoid exponential list size
  - values truncated are closely approximated by the values staying in the list
- (1+ \( \epsilon \)) approximation rate, for a given \( \epsilon \)
- \( \epsilon \) is a parameter of the TRIM function