1 Week 1-Reasoning and asymptotic analysis

1.1 Correctness of algorithm

1.1.1 Correctness of iterative algorithm

A **loop invariant** is:

- true at the beginning of an iteration
- remains true at the beginning of the next iteration

To proof the correctness of an iterative algorithm, We need to show 3 things:

- **Initialization:** The invariant is true before the first iteration of the loop.
- **Maintenance:** If the invariant is true before an iteration, it remains true before the next iteration.
- **Termination:** When the algorithm terminates, the invariant provides a useful property for showing correctness.

1.1.2 Correctness of recursive algorithm

To proof the correctness of an iterative algorithm:

- Use strong induction
- Prove base cases
- Show algorithm works correctly assuming algorithm works correctly for smaller cases.

1.2 Efficiency

Asymptotic Analysis is a method of describing the limiting behavior. Asymptotic notations:

- O-notation (BIG-O)
  (upper bound)
  - $O(g(n)) = \{f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
- Θ-notation (Theta)
  (tight bound)
  - $\Theta(g(n)) = \{f(n)$: there exist constants $c_1 > 0$, $c_2 > 0$, $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
- Ω-notation (BIG-Omega)
  (lower bound)
  - $\Omega(g(n)) = \{f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$
- o-notation (small-o)
  (tight upper bound)
  - $o(g(n)) = \{f(n)$: for any constant $c > 0$, there is a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$
- ω-notation (small-omega)
  (lower bound)
  - $\omega(g(n)) = \{f(n)$: for any constant $c > 0$, there is a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

2 Week 2-Recurrence and Mater Theorem

2.1 Properties of functions (MATHS)

2.1.1 Exponential

$$a^{-1} = 1/a$$
$$(a^m)^n = a^{mn}$$
$$a^m a^n = a^{m+n}$$
$$e^x \geq 1 + x$$

2.1.2 Logarithm

$$\lg n = \log_2 n$$
$$\ln n = \log_e n$$
$$\lg^k n = (\lg n)^k$$
$$lg\, lg\, n = lg(lg\, n))$$
$$a = b^{\log_b a}$$
$$\log_c(ab) = \log_c a + \log_c b$$
$$\log_b a^n = n \log_b a$$
$$\log_b a = \frac{\log_c a}{\log_c b}$$
$$\log_b(1/a) = -\log_b a$$
$$\log_b a = \frac{1}{\log_a b}$$
$$a^{\log_b c} = c^{\log_b a}$$

Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right)$$
$$\log(n!) = \theta(n \lg n)$$

2.1.3 Summation

Arithmetic Series

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + ...n$$
$$= \frac{1}{1} n(n+1) = \Theta(n^2)$$

Geometric Series

$$\sum_{k=1}^{n} x^k = 1 + x^2 + x^3 + ...x^n$$
$$= \frac{x^{n+1} - 1}{x - 1}$$
$$\sum_{k=1}^{\infty} x^k = \frac{1}{1-x} \text{when } | x | < 1$$

Harmonic Series

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$
$$= \sum_{k=1}^{n} \frac{1}{k}$$
$$= \ln n + O(1)$$

Telescoping Series

For any sequence $a_0, a_1, ..., a_n$

$$\sum_{k=0}^{n-1}(a_k - a_{k+1}) = \begin{array}{c}(a_0 + \cancel{a_1}) + \\ (\cancel{a_1} + \cancel{a_2}) + \\ (\cancel{a_2} + \cancel{a_3}) + \\ ... \\ (\cancel{a_{n-1}} + a_n)\end{array} = a_0 - a_n$$

2.1.4 Limit

Assume $f(n), g(n) > 0$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \to f(n) = o(g(n))$$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \to f(n) = O(g(n))$$

$$0 < \lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \to f(n) = \Theta(g(n))$$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \to f(n) = \Omega(g(n))$$

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \to f(n) = \omega(g(n))$$

L'Hopital's Rule

$$\lim_{x\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{x\to\infty}\left(\frac{f'(n)}{g'(n)}\right)$$

2.2 Properties of big-O

Transitivity

$$\begin{aligned} f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \\ \Rightarrow f(n) = \Theta(h(n)) \\ f(n) = O(g(n)) \wedge g(n) = O(h(n)) \\ \Rightarrow f(n) = O(h(n)) \\ f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \\ \Rightarrow f(n) = \Omega(h(n)) \\ f(n) = o(g(n)) \wedge g(n) = o(h(n)) \\ \Rightarrow f(n) = o(h(n)) \\ f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \\ \Rightarrow f(n) = \omega(h(n)) \end{aligned}$$

Reflexivity

$$\begin{aligned} f(n) = \Theta(f(n)) \\ f(n) = O(f(n)) \\ f(n) = \Omega(f(n)) \end{aligned}$$

Symmetry

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

Complementary

$$\begin{aligned} f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n)) \\ f(n) = o(g(n)) \text{ iff } g(n) = \omega(f(n)) \end{aligned}$$

2.3 Solving Recurrences

2.3.1 Substitution Method

Guess the time complexity and verify that it is correct by induction

2.3.2 Telescoping Method

Expand out the recurrence, until the base case then add up

2.3.3 Recursion Tree

Draw out the recurrence in the form of a tree

2.3.4 Master method

Master theorem applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1, b > 1$ and f is asymptotically positive
When comparing $f(n)$ and $n^{\log_b a}$ There are three cases to master theorem

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$. Then $T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}\log^k n)$ for some constant $k \geq 0$. Then $f(n) = \Theta(n^{\log_b a}\log^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ **AND** $af(n/b) \leq cf(n)$ for some constant $c < 1$. Then $T(n) = \Theta(f(n))$

3 Week 3-Divide and Conquer

1. **Divide** the problem (instance) into subproblems.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** subproblem solutions.

Some examples:

- **Binary Search**
  1. **Divide** Check the middle element.
  2. **Conquer** Recursively check 1 sub array.
  3. **Combine** Trivial.

$$T(n) = 1T(n/2) + \Theta(1)$$
$$T(n) = \Theta(\lg n)$$

- **Powering**
  1. **Divide** Trivial.
  2. **Conquer** Recursively compute $f(\lfloor(n/2)\rfloor)$.
  3. **Combine** $f(n) = f(\lfloor(n/2)\rfloor)^2$ if n i even;
     $f(n) = f(1) * f(\lfloor(n/2)\rfloor)^2$ if n is odd.

$$T(n) = 1T(n/2) + \Theta(1)$$
$$T(n) = \Theta(\lg n)$$

- **Fibonacci Number**

Similar to powering a number

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

- **Matrix Multiplication** $A * B$
  Stassen's Algorithm
  1. **Divide** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using + and –.
  2. **Conquer** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
  3. **Combine** Form C using + and – on the seven $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7T(n/2) + \Theta(n^2)$$
$$T(n) = \Theta(n^{\lg 7})$$

- **VLSI Layout**
  – Embed a complete binary tree with n leaves in a grid using minimal area.

$$W(n) = \Theta(\sqrt{n})$$

4 Week 4-Sorting

4.1 Decision Tree

A tree-like model:

- Every node is a comparison
- Every branch represents the output
- Every label represents a class label(decision after all comparison)
- Each leaf is a permutation of the possible ordering(roughly speaking)
- Worst case running time = height of tree
- Height = log(no. of arangement)

4.2 Classification of Sorting Algorithms

4.2.1 Running time

- $O(n^2)$
- $O(n \log n)$

4.2.2 In-place

- Uses very little additional memory, beyond that used by the data. Usualy $O(1)$
- Insertion Sort
- Quicksort ($O(\lg n)$ additional memory with proper implementation)

4.2.3 Stable

- The original order of equal elements is preserved after sorting
- Insertion Sort
- Merge Sort

4.2.4 Comparison or not

- Comparison-based: Compares the element. $\Omega(n \log n)$ is the lower bound for comparison based sorting.
- Non-Comparison-based: no comparison between elements, Linear time.

4.3 Comparison based sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

4.4 Linear time sorting

- Counting Sort
- Radix Sort

5 Week 5-Randomized Algorithms

An algorithm is called **randomized** if its behavior is determined not only by its input but also by values produced by a random-number generator

5.1 Types of Randomized Algorithms

1. **Monte Carlo Algorithm**: Randomized algorithm that gives the correct answer with probability 1 - o(1) ("high probability"), but runtime bound holds deterministically
   - finding π by randomly sampling n (x,y) and count fractions satisfying $x^2 + y^2 \leq 1$ then multiply by 4 to get an estimate to π
   - run is $\Theta(n)$ but only approximates
2. **Las Vegas Algorithm**: Randomized algorithm that always gives the correct answer, but the runtime bounds depend on the random numbers.
   - Randomized Search
   - worst case $\Theta(n)$, expected case $\Theta(\log n)$ depends on random numbers

5.2 Average vs Expected running time

- **average running time**: For non-randomized algorithms that depend on input, if we know the distribution of the input we can find the average running time
- **expected running time**: For randomized algorithms, it depends on random number generator, even for the same input. The "Average" running time over all possible numbers is the expected running time

5.3 Probability

- A and B are not **mutually exclusive** (i.e. $A \cap B \neq \emptyset$): $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\} - Pr\{A \cap B\}$
- Two events are **independent** if $Pr[A \cap B] = Pr[A] \cdot Pr[B]$
- The **Conditional Probability** of an event A given event B is

$$Pr[A|B] = \frac{Pr[A \cup B]}{Pr[B]}$$

whenever $Pr[B] \neq 0$

- **Bayes Theorem**

$$Pr\{A|B\}] = \frac{Pr\{A\}Pr\{B|A\}}{Pr\{B\}}$$
$$= \frac{Pr\{A\}Pr\{B|A\}}{Pr\{A\}Pr\{B|A\} + Pr\{\bar{A}\}Pr\{B|\bar{A}\}}$$

- **expectation** or **mean** of a random variable X is

$$E[X] = \Sigma_x x \cdot Pr[X = x]$$

- **Linearity of Expectations**

$$E[X + Y] = E[X] + E[Y]$$
$$E[aX] = aE[X]$$

- **Expectation of Product** if X and Y are independent

$$E[XY] = E[X]E[Y]$$

- **Bernoulli Trial**: an instance of a Bernoulli trial has probability p of success and probability $1-p = q$ of failure
- **Geometric Distribution**: suppose we have a sequence of independent Bernoulli trial, each with probability p of success. let X be the number of trials needed to obtain success for the first time . Then, X follows the geometric distribution:

$$Pr[X = k] = q^(k-1)p$$
$$E[X] = \frac{1}{p}$$

- **Binomial Distribution**: let X be the number of successes in n Bernoulli trials. Then X follows the binomial distribution

$$Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}$$
$$E[X] = n[$$

5.4 Indicator Random Variable Method

Indicator random variable for an event A:

$$I[A] = \begin{cases} 1, & A \text{ occures} \\ 0, & \text{otherwise} \end{cases}$$

$$E[I[A]] = Pr[A]$$

6 Week 6-Order Statistics

Given an unsorted list, we want to find the element that has rank i.

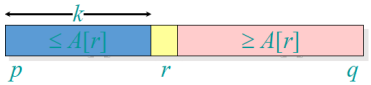
- $i = 1$: minimum
- $i = n$: maximum
- $\lfloor (n+1/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: median

Normally achieved by first sorting the list then reporting the element at index i: $\Theta(n \lg n)$

6.1 finding rank-i element

6.1.1 Randomized Divide and Conquer

```
RAND-SELECT(A[p..q], i)   ▷ ith smallest of A[p..q]
  if p = q then return A[p]
  r ← RAND-PARTITION(A[p..q])
  k ← r − p + 1           ▷ k = rank(A[r])
  if i = k then return A[r]
  if i < k
     then return RAND-SELECT(A[p..r−1], i)
     else return RAND-SELECT(A[r+1..q], i − k)
```



The idea is to randomly select a pivot then check the rank of the pivot, if the element we are looking for is int the left, recursively do it in the left sublist, if it is in the right, do so on the right

6.1.2 Worst case linear time

```
SELECT(i, n)
1. Divide the n elements into groups of 5. Find
   the median of each 5-element group by rote.
2. Recursively SELECT the median x of the ⌊n/5⌋
   group medians to be the pivot.
3. Partition around the pivot x. Let k = rank(x).
4. if i = k then return x
   elseif i < k
      then recursively SELECT the ith
           smallest element in the lower part
      else recursively SELECT the (i–k)th
           smallest element in the upper part
```

Same as RAND-SELECT

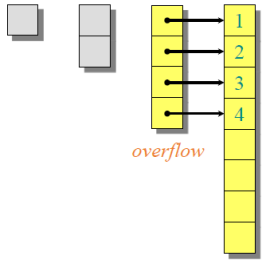
The idea is that we generate a good pivot each time, instead of randomly. The median of the median of the groups has at least $3n/10$ elements greater or less than it.

7 Week 7-Amortized Analysis

Amortized analysisis a strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. Without the use of probability.

Consider the following case: We have a dynamic array initially of size 1. When we insert, if the list is full (overflow) we copy over to a new array with twice the size

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

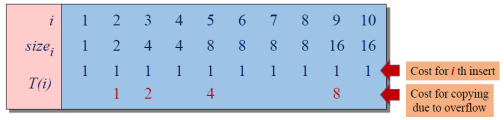


Below are methods to perform amortized analysis

7.1 Aggregate method

The idea is that we use maths
Based on the problem above. Let $t(i) =$ the cost of the ith insertion

$$= \begin{cases} i & \text{if i-1} is a exactpower of 2 \\ 1 & \text{otherwise} \end{cases}$$



Cost of n insertions $= \sum_{i=1}^{n} t(i)$

$$\leq n + \Sigma_{j=0}^{\log(n-1)} 2^j$$
$$\leq 3n$$

Thus the average cost of each insertion in the dynamic array is $O(n)/n = O(1)$

7.2 Accounting method

The idea is to impose an extra charge on inexpensive operations and use it to pay for expensive operations later on. Excess money goes to the bank to be used for future operations. **Need to prove** that bank will never go negative.
Some observations in the problem above

- at insertion i overflow happens, thus next overflow at insertion $2i$
- between then there are i insertions and we need to copy over $2i$ items.

Thus to find amortized cost

- Each insertion is charged \$3
  - \$1 is for the current insertion
  - \$2 is stored to handle overflow
- In the case of no overflow, insert for \$1 and store \$2
- In the case of overflow
  1. Between this overflow and the previous overflow we have i
  2. So we have $\$2 * i$ money in the bank,, use this to copy over, leaving 0 in the bank
  3. The new item that caused the overflow is then inserted to the bigger array normally

7.3 Potential method

ϕ: Potential function associated with the algorithm/data structure $\phi(i)$: Potential at the end of the ith operation
Important conditions to be fulfilled by ϕ

- $\phi(0) = 0$
- $\phi(i) \geq 0$ for all i

Amortized cost of ith operation
=Actual cost of ith operation + $(\Delta\phi(i))$
=cost of ith operation + $(\phi(i) - \phi(i-1))$

Amortized cost of n operations
$=\Sigma_i$Amortized cost of ith operation
=Actual cost of n operations + $\phi(n)$
$\geq$ Actual cost of n operations
Need to find a suitable Potential function ϕ, so that for the costly operation $\Delta\phi_i$ is negative such that is nullifies or reduces the effect of the actual cost.
try to find what is decreasing in the expensive operation

For the example above
$\phi(i) = 2i - \text{size}(T)$

1. ith insertion causes overflow. Actual cost= i. size=$i-1$ before new creation
   - $\phi(i-1) = i-1$
   - $phi(i) = 2*i - 2*(i-1) = 2$
   - $\Delta\phi_i = 3 - i$
2. ith insertion does not cause overflow. Actual cost = 1. size = l
   - $\phi(i-1) = 2(i-1) - l$
   - $\phi(i) = 2i - l$
   - $\Delta\phi_i = 2$

Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1	2	3
Case 2: when table is already full	i	$3 - i$	3

8 Week 8-Dynamic Programming

Suppose we have a recursive solution
overall there are only a polynomial number of subproblems
And there is a huge overlap among the subproblems, the recursive algorithm takes exponential time because it solves the same problem many times)
So we compute the recursive solution iteratively in a bottom up manner, and memoize / remember past solutions to avoid wastage of computation

9 Week 9-Greedy Algorithm

At each step of solving the algorithm, we need only solve one (greedy) sub problem each step.

Given a decision problem P
instance A is of size n of problem P
we perform a greedy step to reduce the problem to
instance A' of size $< n$ of problem P
To prove that the greedy step is correct

1. Try to establish a relation between OPT(A) and OPT(A')
2. Try to prove the relation formally by
   - deriving a (not necessarily optimal) solution of A from OPT(A')
   - deriving a (not necessarily optimal) solution of A' from OPT(A)
3. If you succeed, this algorithm is correct

OPT denotes optimal algorithm to solve

10 Week 10-Intractability

To determine how hard a problem is, we use the idea of Reduction
A $\Rightarrow$ B
We take a general instance of the Problem A transform it to a specific problem B We can use some algorithm M to solve B, then use solution to solve A
if B is easy $\rightarrow$ A is easy if A is hard $\rightarrow$ B is hard
We focus on if A is hard $\rightarrow$ B is hard

$$A \leq_q B$$

make sure reduction is very fast, or else also no point reduction must in polynomial time

10.1 Optimization vs Decision

Optimization:

- give th min/max
- satisfy constraints
- ex: visit each city with min cost

Decision:

- A problem instance, check is it solvable or not
- ex: can i visit every city within cost k

10.2 Reductions between Decision Problems

Given two decision problem A and B, a polynomial time reduction from A to B, denoted $A \leq_p B$, is a transformation from instance α of A to instance β of B such that:

- Transformation must run in polynomial time in the size of α. Word input encoding length
- α is a YES-instance for A if and only if β is a YES-instance for B
  - yes for B $\rightarrow$ yes for A
  - yes for A $\rightarrow$ yes for B

11 Week 11-NP completeness

Complexity grouping:

- P - (polynomial) - The set of decision problems which have efficient poly-time algorithm
- NP - (Non-deterministic polynomial time) - The set of all decision problems which have efficient certifier
- NP-complete: A problem X in NP class is NP-complete if
  for every $A \in$ NP, $A \leq_p X$
- NP-Hard: A in NP $\leq_p$ B in NP hard. IfX is not known to be in NP, then we just say X is in NP-hard

How to show that a problem is NP-complete:

1. Let X be the problem we wish to prove is in NP-complete
2. Show that $X \in$ NP
3. Pick a problem A which is already known to be in NP-complete
4. Show that $A \leq_p X$

12 Week 12-approximation

Sometimes a problem is just too hard to solve, so we approximate the solution.
For an optimization problem, find a solution that is nearly optimal in cost

12.1 Approximation Ratio

Let C^* be the optimal algorithm and C be the cost of the solution given by an approximation algorithm.
An approximation algorithm A has an approximation ratio $\rho(n)$ if:

- for minimization problem
$$\frac{C}{C^*} \leq \rho(n), \rho(n) \geq 1$$
- for maximization problem
$$\frac{C}{C^*} \geq \rho(n), \rho(n) \leq 1$$

Here Cost refers to the maximization or minimization result C

12.2 Analyzing Approximation Algorithms

- **Analyze a Heuristic**: A heuristic is a procedure that does not always produce the optimal answer, but we can show that it is not too bad. Compare heuristic with an optimal solution to find approximation ratio
- **Solve an Linear Programming relaxation**: Not Covered

13 Problem Definitions and Algorithms

13.1 3-SAT

is **NP-Complete**
Definition: SAT where each clause contains exactly 3 literals corresponding to different variables.

$$(\bar{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x_2} \vee x_3)$$

Optimization Version:
Decision Version: Given a 3-SAT, does there exist a satisfying assignment

13.2 Traveling Salesman Problem

is **NP-Hard**
Definition: Given an undirected graph $G = (V, E)$, find a cycle that does through each vertex once and returns to the starting vertex.
Optimization Version: What is the shortest total distance of the cycle
Decision Version: Does there exist a cycle of total distance k

13.3 Independent Set

Definition: Given an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is said to be an independent set if
For each $u, v \in X, (u, v) \notin E$
There is no edge connecting any two vertex in the independent set
Optimization Version: Compute Independent Set of largest size
Decision Version: Does there exist an independent set of size $> k$

13.4 Vertex Cover

is **NP-Complete**
Definition: Given a graph $G = (V, E)$, a vertex cover V' is a subset of V such that $\forall (u, v) \in E : u \in V' \vee v \in V'$
every edge has at least one end point in the vertex cover
Optimization Version: What is the smallest size of a vertex cover
Decision Version: Does there exist a vertex cover of size k

13.5 Partition

Definition: Given a set of positive integers set S, can the set be partitioned into two sets of equal total sum.

13.6 Hamiltonian Cycle

is **NP**
Definition: Given a graph G, Decide whether there is a simple cycle that visits each vertex exactly once.

13.7 Max-Clique

is **NP-Complete**
Definition:Given a graph $G = (V, E)$, does there exist a subset V' of V, such that all vertices in V' are adjacent to each other
Optimization Version: What is the largest max clique
Decision Version: Does there exist a max clique of size k

13.8 Knapsack

Definition: Given n items described by non-negative integer pairs (w_i, v_i), capacity W and value V. w_i denotes the weight of an item, v_i denotes the value of the item.
Optimization Version: What is the maximum value a subset of item of total weight at most W
Decision Version: Is there a subset of item of total weight at most W and total value at least V