Bit Algo
START

Algorytmy zachłanne

Wady programowania dynamicznego

- **ostry warunek overlapping subproblems** - często mamy spełnione pozostałe 3 (recursive def., overlapping subproblems, optimal substructure), a ten nie
- **nadal trzeba wszystko policzyć** - programowanie dynamiczne "przycina" drzewo rekurencji (każdą wartość liczymy tylko raz), ale dalej wymaga policzenia bardzo wielu wartości (często nawet wszystkich możliwych)

Algorytmy zachłanne

- **idea:** za każdym razem dokonujemy **aktualnie** optymalnego wyboru, zachłannie bierzemy to, co w danej chwili uważamy za najlepsze
- **optymalizacja obliczeń -** obliczamy **tylko** wybrany zachłannie krok do rozwiązania, nie obliczamy pozostałych możliwości (w przeciwieństwie do programowania dynamicznego)
- **podobne do programowania dynamicznego** - też ma jego 3 warunki, ale zamiast overlapping subproblems ma warunek **greedy-choice property**
- **zwykle top-down** - bardziej naturalne dla tych algorytmów, bo wybieramy zachłannie najlepsze rozwiązanie, a dla reszty wywołujemy się rekurencyjnie
- **prawie zawsze można je zastąpić programowaniem dynamicznym** - tyle, że algorytmy zachłanne (wtedy, kiedy da się ich użyć) są szybsze, więc je preferujemy

Algorytmy zachłanne - warunki

1. **Recursive definition**
2. **(almost) Independent subproblems** - podproblemy **mogą** zależeć od poprzednio wybranych rozwiązań podproblemów, ale **nie** mogą zależeć od rozwiązań innych podproblemów
3. **Optimal substructure**
4. **Greedy-choice property** - wybór zachłanny (elementu, który stanowi optymalne rozwiązanie aktualnego podproblemu) **musi** prowadzić do rozwiązania optymalnego całości

Ostatni warunek jest najważniejszy - to właśnie on pozwala na wykorzystanie algorytmów zachłannych. Jest stosunkowo **rzadki** i trzeba go **udowodnić**.

Problem wyboru aktywności

Dana jest tablica n aktywności, $S = [a_1, a_2, \dots , a_n]$. Każda aktywność a_i ma **start_time** s_i i **finish_time** f_i (l. naturalne), gdzie $0 \le s_i < f_i < \infty$. Aktywność odbywa się w przedziale czasowym $[s_i, f_i)$.

Wszystkie aktywności (np. wykłady) wymagają tego samego zasobu, np. sali, w której można nagrywać wykłady podczas kwarantanny. Aktywności nazywamy **kompatybilnymi**, gdy na siebie nie zachodzą (tzn. f_i jednej aktywności jest co najwyżej równy s_i drugiej aktywności).

W **problemie wyboru aktywności** chcemy wybrać **jak najwięcej kompatybilnych aktywności**, tzn. największy podzbiór kompatybilnych aktywności.

Założenie: aktywności są posortowane monotonicznie rosnąco według ich finish_time, tzn.:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$$

Problem wyboru aktywności

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Pewien wybór aktywności: $\{a_3, a_9, a_{11}\}$

Optymalny wybór: $\{a_1, a_4, a_8, a_{11}\}$

Inny optymalny wybór: $\{a_2, a_4, a_9, a_{11}\}$

Cormen i indeksowanie od 1

Dana tablica aktywności to S = $[a_1, a_2, \dots, a_n]$ - indeksowana od 1; Python indeksuje od 0.

Rozwiązanie:

Robimy tablicę rozmiaru (n+1), a zerowy element ignorujemy. Co prawda za chwilę zobaczymy, że warto użyć indeksu 0, ale nie będzie on używany do faktycznego liczenia czegokolwiek.

P. w. a. - dowód optimal substructure

S_{ij} - **zbiór** aktywności, które zaczynają się **po** zakończeniu a_i i **przed** rozpoczęciem a_j

A_{ij} - **największy** zbiór kompatybilnych aktywności dla S_{ij}

a_k - pewna wybrana aktywność

Gdy wybierzemy a_k do A_{ij}, to dostaniemy: $A_{ij} = A_{ik} + a_k + A_{kj} \Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

A_{ij} **musi** zawierać optymalne rozwiązania S_{ik} i S_{kj}. Gdyby istniało np. A_{kj}' takie, że $|A_{kj}'| > |A_{kj}|$, to wtedy mielibyśmy: $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ (sprzeczność).

Z powyższego wynika, że optymalne rozwiązanie A_{ij} **wymaga** optymalnych rozwiązań podproblemów A_{ik} i A_{kj}.

P. w. a. - wykorzystanie dowodu

Skoro udowodniliśmy, że $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$, to mamy prosty wzór rekurencyjny na nasze poszukiwane optymalne rozwiązanie.

Oznaczmy optymalne rozwiązanie dla S_{ij} przez c[i, j], mamy wtedy:

$$c[i, j] = c[i, k] + c[k, j] + 1$$

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

P. w. a. - da się lepiej!

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

W powyższym wzorze obliczenie max(...) jest czasochłonne - trzeba w końcu policzyć dla wszystkich a_k.

Zależy nam na tym, żeby wybrać **jak najwięcej aktywności**. Można by więc wybrać najszybciej kończącą się aktywność - tak, aby zostawić jak najwięcej czasu pozostałym.

Wykorzystujemy informację z zadania - skoro mamy dane aktywności posortowane według finish_time, to wystarczy brać “pierwszą z brzegu” aktywność.

W ten sposób **nie musimy przejmować się pozostałymi możliwościami**.

P. w. a. - dowód greedy-choice property

S_k - pewien podproblem; A_k - największy podzbiór kompatybilnych aktywności w S_k

a_m - aktywność w S_k o **najmniejszym** finish_time (ją chcemy zachłannie wybrać)

a_j - aktywność w A_k o **najmniejszym** finish_time (ją musimy wybrać, żeby było optymalnie)

Teza: a_m należy do A_k (= można je zachłannie wybrać)

Dowód: Jeżeli $a_j = a_m$, to gotowe, bo a_m należy do największego podzbioru A_k (= jest optymalne).

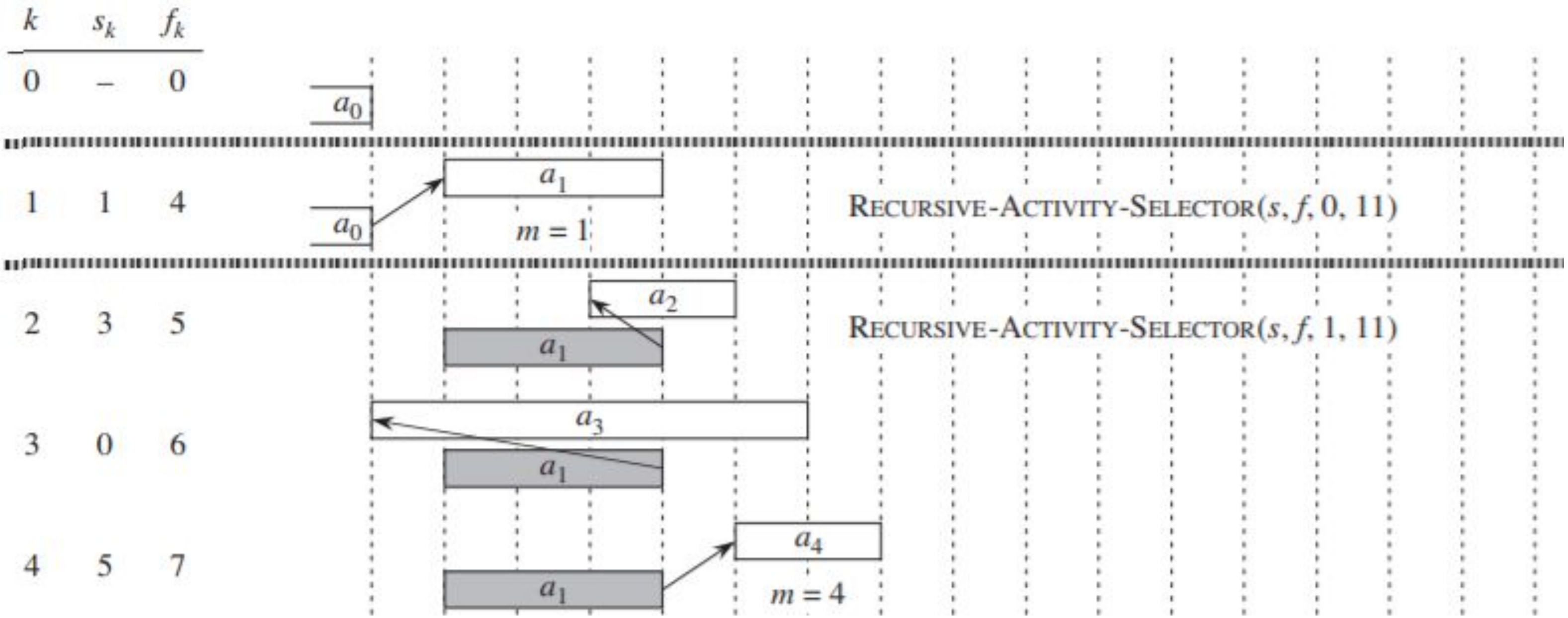
W p. p. ($a_j \neq a_m$) stwórzmy nowy zbiór: $A'_k = A_k - \{a_j\} \cup \{a_m\}$ (zamieniliśmy a_j na a_m). Skoro a_j kończyło się najszybciej w A_k, a a_m kończy się przed a_j (z definicji), to a_m jest kompatybilne z A_k'. Jako że $|A_k| = |A_k'|$, to A_k' to największy podzbiór kompatybilnych aktywności w S_k, więc jest optymalnym rozwiązaniem.

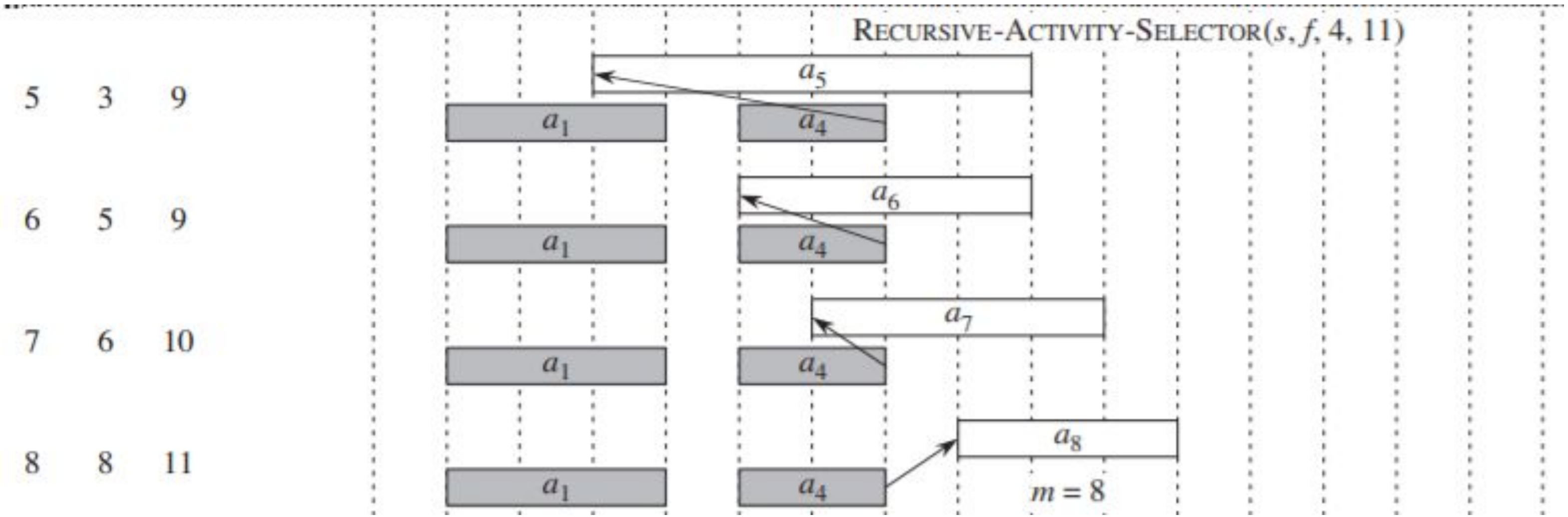
P. w. a. - zachłanny algorytm rekurencyjny

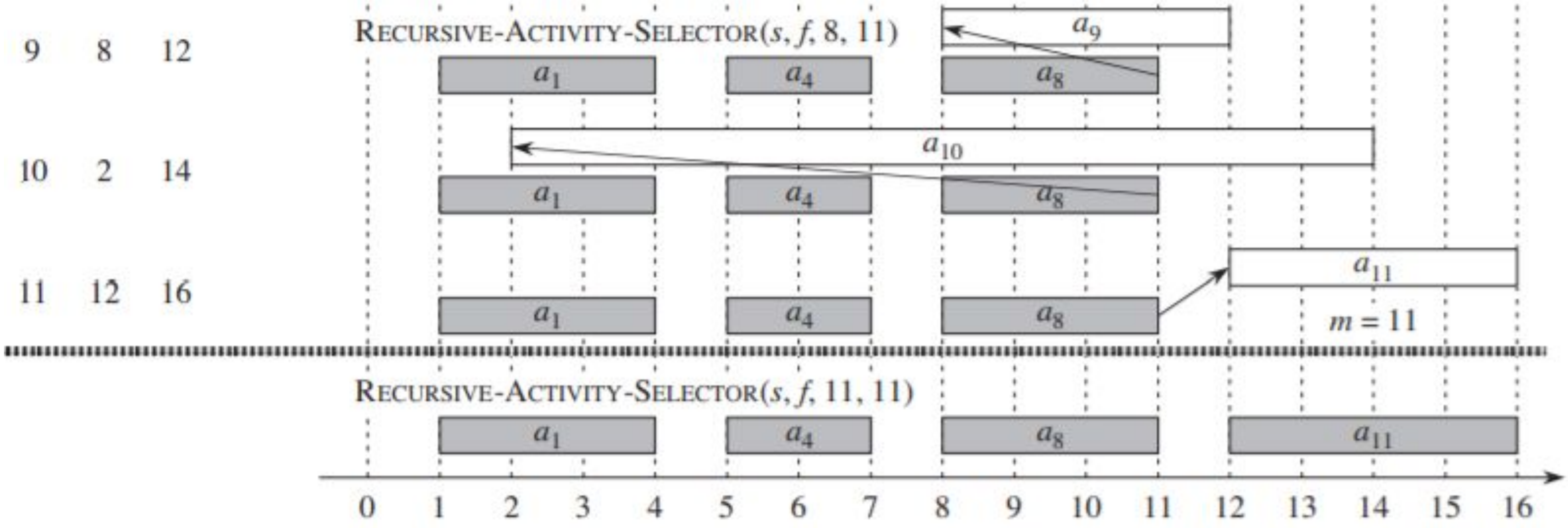
Możemy zatem użyć algorytmu, który po prostu wybiera najszybciej kończącą się aktywność jako kolejną do rozwiązania. Uwaga - wywołujemy `Recursive-Activity-Selector(s, f, 0, n)` . W Pythonie zamiast tablic s i f mamy jedną tablicę obiektów klasy Activity z atrybutami start_time i finish_time.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

Bit Algo START




RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
5 3 9
6 5 9
7 6 10
8 8 11
a5
a1
a4
a6
a7
a8
m = 8


9 8 12
10 2 14
11 12 16
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
a9
a1
a4
a8
a10
a11
m = 11
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

P. w. a. - zachłanny algorytm iteracyjny

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

→

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

O(n)

Algorytmy zachłanne - podsumowanie teorii

- **idea:** bierzemy rozwiązanie, które w tej chwili uważamy za optymalne
- **szybsze od programowania dynamicznego** - nie rozważamy wszystkich możliwości, tylko szybko bierzemy to, co uważamy w danej chwili za optymalne
- **warunek greedy-choice property** - musimy udowodnić, że wybór zachłanny lokalnie optymalnego rozwiązania doprowadzi nas do rozwiązania optymalnego całego problemu
- **rzadsze od programowania dynamicznego** - ze względu na rzadkość powyższej wartości
- **łatwa konstrukcja rozwiązania** - zawsze wiemy, co bierzemy do rozwiązania, więc na koniec wystarczy zwrócić wszystko to, co zachłannie wybraliśmy

Ciągły problem plecakowy

Opis problemu: Złodziej dysponuje plecakiem o znanej pojemności W. Przed sobą widzi szereg produktów, z których każdy ma pewną określoną cenę na jednostkę masy. Należy podać jaką część masy każdego produktu musi zabrać, tak aby nie przekroczył dostępnej pojemności plecaka i całkowita cena była maksymalna.

Przekładając na zrozumiały język: Dana jest liczba rzeczywista W, oraz lista par (v, w) w której v oznacza jednostkową cenę i-tego produktu, a w dostępną masę. Dla każdego elementu tablicy należy podać takie x, że suma x*(v/w) dla wszystkich elementów jest maksymalna.

Ciągły problem plecakowy - heurystyka

Dla uprzednio zdefiniowanego problemu możemy zaproponować następującą heurystykę:

Posortuj malejąco produkty po cenie na jednostkę masy. Następnie idąc od początku tablicy zabierz każdego produktu tyle ile się da, ale tak, żeby nie wziąć więcej niż jest dostępne i nie przekroczyć pojemności plecaka.

Ciągły problem plecakowy - ilustracja

objects	1	2	3	4	5	6	7
profits	10	5	15	7	6	18	3
weights	2	3	5	7	1	4	1
p/w	5	1.6	3	1	6	4.5	3

Ciągły problem plecakowy - dowód

Niech rozwiązanie wygenerowane przez algorytm będzie dane jako: G = (x1, x2, …, xn), a rozwiązanie optymalne O(y1, y2, …, yn). Niech i będzie pierwszą pozycją, na której rozwiązania się różnią. Oznaczamy d = xi-yi. Należy zauważyć, że waga przedmiotów zabrana w rozwiązaniach G i O musi być taka sama!. Teraz tworzymy nowe rozwiązanie O’. W tym rozwiązaniu ułamki do i -1 są takie same jak w O, na miejsce i wstawiamy xi, natomiast pozostałe modyfikujemy, tak aby waga zmniejszyła się o d*wi. O’ nie może mieć mniejszej ceny niż O, bo zwiększaliśmy wagę najcenniejszego przedmiotu, ale nie może też mieć mniejszej niż O, ponieważ o było optymalne. Zatem O’ jest jednym z optymalnych rozwiązań. Zauważając, że powtarzając ten krok i za każdym razem podstawiając O = O’, konstruujemy rozwiązanie G. Zatem G też jest optymalne.

Ciągły problem plecakowy - pseudokod

```
def FractionalKnapsack(array, W):
    sort(array, key = first)
    currW = 0.0
    cost = 0.0
    for i in array:
        if(currW + i[1] <= W):
            currW+=i[1]
            cost += i[1]*i[0]
        else:
            cost+=(W-currW)*i[0]
    return cost
```

Kodowanie Huffmana

Opis problemu: Mając dany alfabet, wraz z listą wag (prawdopodobieństw) dla każdego symbolu, stworzyć system kodowania binarnego, tak aby oczekiwana długość kodu była minimalna.

Lista wag = (w1, w2, w3,.., wn)

Lista kodów binarnych dla każdego znaku = (c1, c2, …, cn)

Chcemy minimalizować sumę: wi*length(c1) po i = 1 do n

Kodowanie Huffmana - propozycja algorytmu

Pierwszym etapem algorytmu będzie stworzenie tzw. drzewa Huffmana. Tworzymy je w następujący sposób:

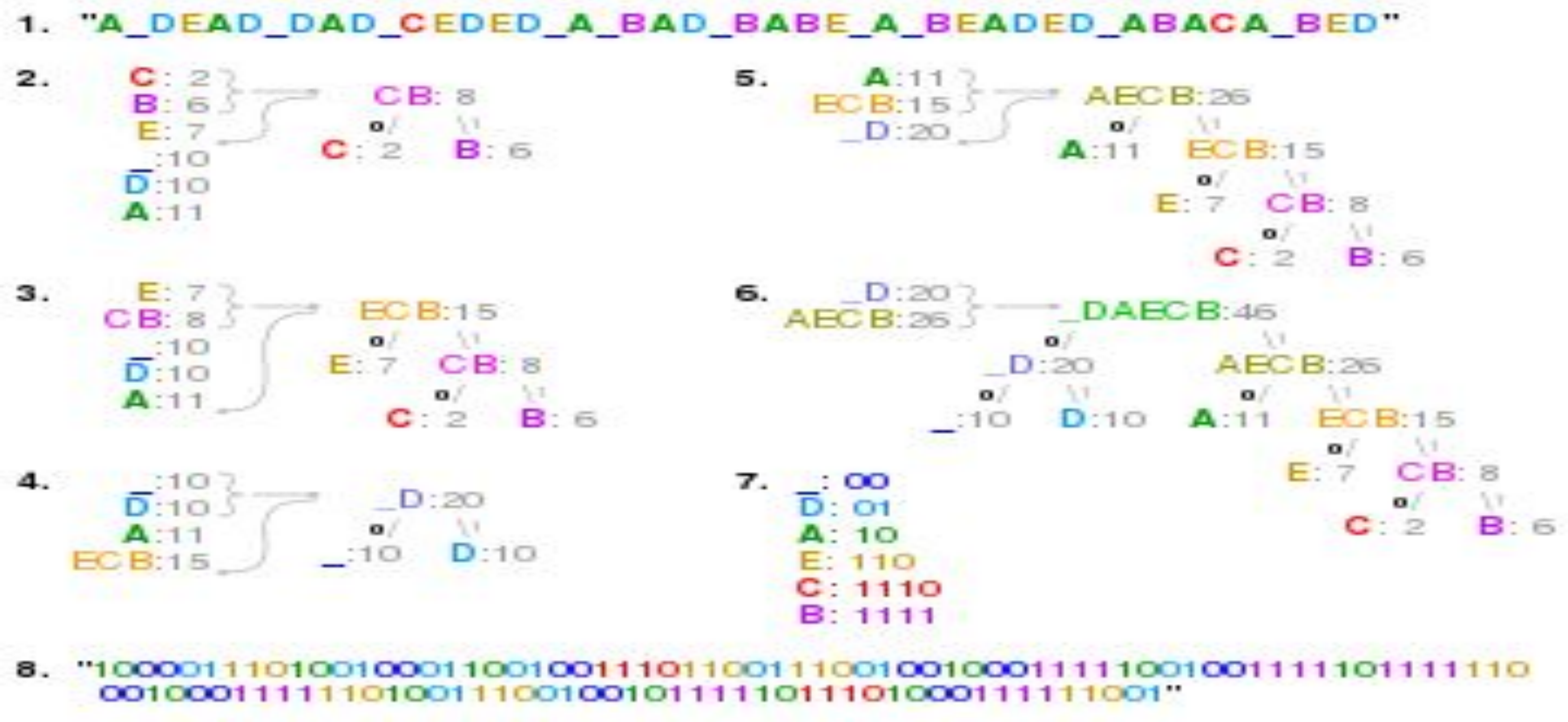
Dla każdej litery utwórz liść, wraz z jej wagą.

Wrzuć wszystkie liście do kopca min, z kluczem sortowania przyjętym jako waga danego liścia.

Powtarzaj:

1. Wyciągnij dwa liście z kopca.
2. Utwórz nowy liść mający wagę równą sumie dwóch wyciągniętych liści i ustaw wyciągnięte l iście jako jego dzieci. Wrzuć nowy liść do kopca

Kodowanie Huffmana - ilustracja



Kodowanie Huffmana - dowód

Algorytm jest w oczywisty sposób poprawny dla alfabetu o rozmiarze n = 2. Załóżmy, że algorytm produkuje poprawne rozwiązanie dla n - 1. Zauważamy ponadto, że długość kodu dla danej literki jest głębokości na jakiej w drzewie znajduje się jej liść.

Zamieniamy dwa liście o najmniejszej wadze na liść jednego symbolu, o wadze równej sumie tych wag. Dostajemy nowy alfabet, dla którego tworzymy optymalne drzewo T. W drzewie T zamieniamy liść nowo utworzonego symbolu na węzeł, z dziećmi będącymi uprzednio zabranymi symbolami.

Dlaczego otrzymaliśmy optymalne drzewo?

http://www.cs.utoronto.ca/~brudno/csc373w09/huffman.pdf

Bit Algo

START