## A black and white logo AI-generated content may be incorrect.

**ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

**ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ**

**Курсов Проект**

**Дисциплина: „Изследване на операциите и приложно програмиране“**

**Тема: Задача за натоварването**

**Изготвил:**

**Леа Петрова Петрова**

**Фак №: 501222029**

**Група: 49**

**IV Курс, ИТИ**

**e-mail:** [**lepetrova@tu-sofia.bg**](mailto:lepetrova@tu-sofia.bg)

## Съдържание:

[Описание на задачата 2](#_Toc216390660)

[Динамично Програмиране като Средство за Оптимизация 3](#_Toc216390661)

[Теоритична част на решението 3](#_Toc216390662)

[Праграма 4](#_Toc216390663)

[Резултати от Изпълнението 6](#_Toc216390664)

[Използвани Източници 7](#_Toc216390665)

# Описание на задачата

Настоящият проект разглежда класическа задача за оптимизация, известна като **Задача за натоварване (Knapsack Problem)** или в конкретика за направения пример, определяща оптималният план за натоварване на кораб с капацитет 10 тона, като се използват контейнери от четири различни типа.

**Целта** на задачата е да се определи оптималното количество контейнери от различни типове, което трябва да бъде натоварено на кораб с ограничен капацитет (), така че **общата стойност на товара да бъде максимална**, като се спазва ограничението за тегло.

* **Дадено:**
  + Капацитет на кораба: тона.
  + Набор от N=4 типа контейнери, всеки с определено тегло () и стойност (). **Дадени са следните параметри за всеки тип контейнер:**

|  |  |  |
| --- | --- | --- |
| **Тип i** | **Тегло ​ (тона)** | **Стойност** ()**​ (ценови единици)** |
| 1 | 2 | 4 |
| 2 | 8 | 10 |
| 3 | 3 | 6 |
| 4 | 4 | 8 |

* + Неограничен брой от всеки тип контейнер!
* **Търси се:** Броят контейнери за всеки тип , който максимизира общата стойност , при условие, че общото тегло **.**

Тъй като тази задача представлява дискретен проблем с множество възможни комбинации, за нейното оптимално решаване се прилага методът на **динамичното програмиране**.

# Динамично Програмиране като Средство за Оптимизация

**Динамичното програмиране (ДП)** е мощен метод за решаване на сложни проблеми чрез разбиването им на по-прости, припокриващи се подзадачи.

* **Принцип на Оптималността**

Методът се базира на **Принципа на Оптималността** на Ричард Белман: Едно оптимално решение има свойството, че независимо какви са предишните решения, останалите решения трябва да формират оптимално решение по отношение на състоянието, получено в резултат на тези предишни решения.

**Приложение в Задачата за Натоварване**

В задачата за натоварване, ДП се прилага, като се дефинира функция на състоянието, която представлява **максималната стойност, която може да бъде получена от типове контейнери** при остатъчен капацитет . Решението на дадена стъпка се свежда до избора на броя контейнери , който дава най-добрия резултат, като се използва вече изчисленото оптимално решение за следващата стъпка .

# Теоритична част на решението

Решението следва **Многостъпков модел на решение**, където всяка стъпка ()

съответства на избора на количеството () на един тип контейнер.

**Дефиниране на Рекурентната Зависимост**

Основната рекурентна формула за решаване на задачата по етапен метод е:

Където:

* е **номерът на стъпката** (типът контейнер, който се разглежда).
* е **състоянието** (остатъчният капацитет на кораба). .
* е **управляващата променлива** (броят контейнери от тип ), като и
* е **гранично условие** (след последния тип контейнер, стойността е нула).

**Процес на Изчисление**

1. **Обратен ход:** Започва се от последната стъпка () и се изчислява за всички възможни състояния до .
2. **Прав ход:** Започва се от началното състояние и се определя оптималният . След това, с помощта на остатъчния капацитет, се определя оптималният , и така нататък до .

# Праграма

Програмата **solveKnapsackStageDP(capacity, items)** реализира описания алгоритъм на JavaScript.

**Структура на Данните**

* **items**: Масив от обекти, съдържащ теглото () и стойността () за всеки тип контейнер.
* **W**: Основният ДП масив. W[i][S] съхранява обект { W: max\_cost, x: optimal\_x }, който представлява максималната стойност и съответното оптимално количество за стъпка при капацитет .

**Ключови Елементи в Кода**

1. **Сортиране (sortedItems)**: Артикулите се сортират в обратен ред по тип (4, 3, 2, 1), за да може цикълът да започне от последната стъпка (), както изисква ДП формулата.
2. **Итеративно Изчисление (for (let k = 0; k < N - 1; k++))**:
   * Този цикъл обхваща стъпки .
   * Вътрешният цикъл обхожда всички състояния .
   * Най-вътрешният цикъл итерира всички възможни бройки , прилагайки рекурентната формула и разчитайки на вече изчислените стойности .
3. **Изчисляване на** : Стъпка се изпълнява отделно, за да се намери окончателната максимална стойност (max\_W) и всички стойности на , които водят до нея.
4. **Възстановяване на Решението**: За всяка оптимална се използва **запазената** информация в масива (полето x) от стъпки , за да се проследи пътят напред и да се открие пълният набор .

**Код ([линк към GitHub repo](https://github.com/lea-vpetrova/IOPP_Uni_Course/blob/main/%D0%9A%D1%83%D1%80%D1%81%D0%BE%D0%B2%20%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82/maxLoadEx.js)):**

function solveKnapsackStageDP(capacity, items) {

  const Q = capacity;

  const N = items.length;

  const sortedItems = items.sort((a, b) => b.type - a.type);

  const W = [];

  for (let k = 0; k < N - 1; k++) {

    const currentItem = sortedItems[k];

    const currentStep = currentItem.type;

    const nextStepIndex = currentStep + 1;

    W[currentStep] = [];

    for (let S = 0; S <= Q; S++) {

      const max\_x = Math.floor(S / currentItem.weight);

      let max\_W = -1;

      let optimal\_x = 0;

      for (let x = 0; x <= max\_x; x++) {

        const remaining\_S = S - currentItem.weight \* x;

        const W\_next\_step = W[nextStepIndex]

          ? W[nextStepIndex][remaining\_S].W

          : 0;

        const current\_W = currentItem.cost \* x + W\_next\_step;

        if (current\_W > max\_W) {

          max\_W = current\_W;

          optimal\_x = x;

        }

      }

      W[currentStep][S] = { W: max\_W, x: optimal\_x };

    }

  }

  const firstItem = sortedItems[N - 1];

  const firstStep = firstItem.type;

  const max\_x1 = Math.floor(Q / firstItem.weight);

  let max\_W = -1;

  let optimal\_x1\_values = [];

  for (let x1 = 0; x1 <= max\_x1; x1++) {

    const remaining\_S = Q - firstItem.weight \* x1;

    const W\_next\_step = W[firstStep + 1][remaining\_S].W;

    const current\_W = firstItem.cost \* x1 + W\_next\_step;

    if (current\_W > max\_W) {

      max\_W = current\_W;

      optimal\_x1\_values = [x1];

    } else if (current\_W === max\_W) {

      optimal\_x1\_values.push(x1);

    }

  }

  const all\_optimal\_solutions = optimal\_x1\_values.map((x1) => {

    let solution = { [firstItem.type]: x1 };

    let current\_S = Q;

    current\_S = current\_S - firstItem.weight \* x1;

    for (let k = N - 2; k >= 0; k--) {

      const currentItem = sortedItems[k];

      const currentStep = currentItem.type;

      const optimal\_x = W[currentStep][current\_S].x;

      solution[currentStep] = optimal\_x;

      current\_S = current\_S - currentItem.weight \* optimal\_x;

    }

    const finalSol = {};

    items.forEach((item) => {

      finalSol[`x${item.type}\*`] = solution[item.type] || 0;

    });

    return finalSol;

  });

  return {

    max\_cost: max\_W,

    solutions: all\_optimal\_solutions,

    Q: Q,

    items: items,

  };

}

const course\_items = [

  { type: 1, weight: 2, cost: 4 },

  { type: 2, weight: 8, cost: 10 },

  { type: 3, weight: 3, cost: 6 },

  { type: 4, weight: 4, cost: 8 },

];

const USER\_CAPACITY = 10;

const finalResult = solveKnapsackStageDP(USER\_CAPACITY, course\_items);

console.log(`РЕЗУЛТАТ: Задача за Натоварване (Динамично Програмиране)`);

console.log(`Общ Капацитет (Q): ${finalResult.Q} тона`);

console.log(`МАСИВНИ ДАННИ: Тегло q\_i / Стойност c\_i `);

finalResult.items.forEach((i) =>

  console.log(`  Контейнер ${i.type}: ${i.weight}т / ${i.cost} ценови единици`)

);

console.log(

  `МАКСИМАЛНА СТОЙНОСТ НА ТОВАРА (W1(10)): ${finalResult.max\_cost} ценови единици`

);

console.log("ОПТИМАЛНИ РЕШЕНИЯ (x\_i\* - брой контейнери):");

finalResult.solutions.forEach((sol, index) => {

  const totalWeight =

    sol["x1\*"] \* 2 + sol["x2\*"] \* 8 + sol["x3\*"] \* 3 + sol["x4\*"] \* 4;

  const solStr = Object.keys(sol)

    .map((key) => `${key.replace("\*", "")}=${sol[key]}`)

    .join(", ");

  console.log(

    `  Решение ${index + 1}: ${solStr} (Общо тегло: ${totalWeight}т)`

  );

});

# Резултати от Изпълнението

A screenshot of a computer

AI-generated content may be incorrect.

**Максимална Стойност**

**Оптимални Разпределения**

Това са валидните решения, които запълват точно **10 тона** и постигат максималната стойност **20 цен. ед.**:

# Използвани Източници

1. **Ръководство по изследването на операциите и приложно програмиране**
2. <https://www.geeksforgeeks.org/dsa/0-1-knapsack-problem-dp-10/>
3. <https://en.wikipedia.org/wiki/Knapsack_problem>
4. <https://web.mit.edu/15.053/www/AMP-Chapter-11.pdf>
5. <https://atcoder.jp/contests/dp/tasks/dp_d>