ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Lớp: CS112.Q11.KHTN

Môn học: Phân Tích Và Thiết Kế Thuật Toán

Luyện Tập Thiết Kế Thuật Toán
Nhóm 6

Nhóm 4:
Lê Văn Thức
Bảo Quý Định Tân

Giảng viên:
Nguyễn Thanh Sơn

1 Phương pháp thiết kế thuật toán

1.1 Subtask 1 ($N \leq 20$):

- **Phương pháp:** Sử dụng phương pháp thiết kế **Branch and bound** và **Transfer and conquer (Instance Simplification)**.
- **Nguyên lý và cách áp dụng:** Bài toán được mô hình hóa dưới dạng tìm kiếm trên cây không gian trạng thái nhị phân. Tại mỗi bước xét công việc thứ i, thuật toán thực hiện hai quyết định:

  1. **Chọn công việc i:** Chỉ thực hiện nhánh này nếu thỏa mãn điều kiện ràng buộc về thời gian: $current\_time + r_i \leq t_i$. Nếu chọn, cập nhật thời gian và lợi nhuận tích lũy.
  2. **Không chọn công việc i:** Bỏ qua và chuyển sang xét công việc $i + 1$.

  Tức ta chỉ chọn công việc nếu có thể thực hiện được để tránh lãng phí thời gian (không tối ưu, làm sinh ra các nhánh duyệt thừa) đây là **Branch and bound**.

  Trước khi thực hiện đệ quy, danh sách công việc được sắp xếp theo hạn chót (Deadline t_i) tăng dần để tối ưu hóa khả năng cắt tỉa nhánh sớm **(Transfer and conquer)**.

1.2 Subtask 2 ($N > 20$)

- **Phương pháp:** Sử dụng phương pháp thiết kế **thuật toán gần đúng**: Metaheuristic - Tìm kiếm Vùng lân cận lớn Thích nghi **Adaptive Large Neighborhood Search (ALNS)** và **Transfer and conquer (Instance Simplification)**.
- **Nguyên lý và cách áp dụng:** ALNS cải thiện lời giải thông qua quá trình lặp đi lặp lại gồm hai giai đoạn chính:

  1. **Phá hủy (Ruin/Destroy):** Loại bỏ một số lượng công việc ra khỏi lịch trình hiện tại để thoát khỏi tối ưu cục bộ.
  2. **Tái tạo (Recreate/Repair):** Chèn lại các công việc chưa được xếp lịch vào các vị trí tốt nhất có thể dựa trên các chiến lược tham lam (Greedy heuristics).

  Tính chất "Thích nghi"(Adaptive) thể hiện ở việc thuật toán tự động điều chỉnh trọng số lựa chọn các chiến lược dựa trên hiệu quả mà chúng mang lại trong quá khứ.

2 Tính phù hợp của phương pháp (0.4 điểm)

2.1 Subtask 1 ($N \leq 20$) - Sử dụng Branch and bound

Lý do phù hợp:

- Với $N \leq 20$, không gian trạng thái tối đa là $2^{20} \approx 10^6$. Đây là kích thước đủ nhỏ để máy tính hiện đại xử lý trong thời gian dưới 1 giây.

- Phương pháp Branch and bound là phương pháp chính xác (Exact Method), đảm bảo duyệt qua tất cả các trường hợp khả thi để tìm ra **nghiệm tối ưu toàn cục (Global Optimum)** và cắt tỉa bớt các nhánh không cần thiết để giảm thời gian duyệt, điều cần thiết để đạt điểm tuyệt đối ở các test nhỏ.

2.2 Subtask 2 ($N > 20$) - Sử dụng Metaheuristic

Lý do phù hợp:

- Bài toán là biến thể của bài toán Xếp ba lô (Knapsack Problem) kết hợp Lập lịch, thuộc lớp **NP-Hard**. Khi N lớn (ví dụ $N = 500$), không gian nghiệm bùng nổ theo hàm mũ, không thể dùng các phương pháp chính xác.
- Metaheuristic cho phép tìm kiếm lời giải *đủ tốt* (near-optimal) trong thời gian giới hạn bằng cách cân bằng giữa Khai thác (Exploitation) và Khám phá (Exploration).

Chi tiết về thuật toán ALNS được chọn:

- **Tên thuật toán:** Adaptive Large Neighborhood Search (ALNS).
- **Tại sao chọn ALNS:** So với Giải thuật Di truyền (GA), ALNS phù hợp hơn cho bài toán có ràng buộc chặt (như deadline). Trong GA, phép lai ghép (crossover) thường tạo ra các lịch trình con vi phạm ràng buộc thời gian (invalid solutions). Ngược lại, ALNS làm việc trên một cá thể và sử dụng cơ chế "Repair"tham lam để chèn lại công việc, giúp dễ dàng kiểm soát tính hợp lệ của lịch trình tại mọi bước.
- **Cấu hình tham số (Hyperparameters):**
  - **Thời gian chạy (Time Limit):** 4.85 giây.
    Lý do: Đặt ngưỡng an toàn (Safety buffer) thấp hơn giới hạn 5 giây của hệ thống chấm để đảm bảo chương trình kết thúc và in kết quả kịp thời, tránh lỗi Time Limit Exceeded (TLE).
  - **Trọng số chiến lược (Strategy Weights):** $[50, 20, 10, 20]$ tương ứng với [Efficiency, Profit, Deadline, Balanced].
    Lý do: Trọng số cho "Efficiency"(p_i/r_i) được đặt cao nhất (50) vì trong bài toán Knapsack, mật độ giá trị thường là chỉ dấu tốt nhất cho nghiệm tối ưu. Cơ chế Adaptive sẽ cộng thêm điểm thưởng (+5) nếu chiến lược nào tìm được nghiệm tốt hơn.
  - **Tỷ lệ Ruin (Ruin Rate):** 30% Restart, 70% Continue.
    Lý do: Có 30% xác suất thuật toán quay lại nghiệm tốt nhất toàn cục để tìm hướng khác, giúp cân bằng giữa việc đào sâu vào nghiệm hiện tại và duy trì nghiệm tốt nhất.
  - **Kích thước phá hủy (Destruction/Ruin Size):** Thay vì sử dụng "Mutation Rate"như trong giải thuật di truyền, ALNS sử dụng kích thước vùng phá hủy để tạo sự đa dạng (Diversification):
    - *Random Scatter:* Có 70%: Xóa ngẫu nhiên $3 \to 12$ công việc.
    - *Burst Removal:* Có 30%: Xóa đoạn liền kề $3 \to 15$ công việc.

    Lý do: Ngưỡng $3 \to 15$ (khoảng $0.6\% \to 3\%$ với $N = 500$) là kích thước đủ lớn để tạo ra các cấu trúc nghiệm mới, nhưng đủ nhỏ để giữ lại các đoạn lịch trình đã được tối ưu tốt trước đó (Intensification).

3 Subtask 1: Phân tích thuật toán

3.0.1 Chiến lược Sắp xếp Định hướng sử dụng phương pháp Transform and Conquer (Earliest Due Date Ordering)

Trước khi bắt đầu đệ quy, danh sách công việc được sắp xếp tăng dần theo Deadline ($t_1 \leq t_2 \leq \cdots \leq t_N$).

- **Mục đích:** Trong các bài toán lập lịch, quy tắc EDD (Earliest Due Date) thường tạo ra các lịch trình hợp lệ dễ dàng hơn.
- **Tác động lên cây tìm kiếm:** Việc đưa các công việc có deadline gấp (nhỏ) lên đầu cây nhị phân giúp thuật toán phát hiện sự vi phạm ràng buộc (infeasibility) ngay ở những tầng nông nhất của cây đệ quy. Nếu các công việc gấp không thể thỏa mãn, nhánh đó sẽ bị cắt bỏ ngay lập tức thay vì phải đợi duyệt đến đáy cây.

3.0.2 Thực hiện back tracking với cơ chế Cắt tỉa theo Tính khả thi (Feasibility Pruning)

Tại mỗi bước đệ quy, trước khi quyết định "Chọn"công việc i, thuật toán kiểm tra điều kiện tiên quyết:

$$current\_time + r_i \leq t_i$$

- Nếu điều kiện này sai, toàn bộ nhánh con phía dưới (bao gồm tất cả các tổ hợp có chứa công việc i tại thời điểm này) sẽ bị loại bỏ hoàn toàn (Pruned).
- Nhờ việc kết hợp với sắp xếp EDD ở trên, cơ chế cắt tỉa này hoạt động cực kỳ hiệu quả, giảm không gian tìm kiếm thực tế xuống thấp hơn nhiều so với 2^{20}.

3.0.3 Xử lý giới hạn Đệ quy (Recursion Limit Handling)

Do Subtask 1 sử dụng đệ quy sâu và mặc định của Python giới hạn độ sâu (thường là 1000), chúng tôi chủ động thiết lập: sys.setrecursionlimit(5000) Điều này đảm bảo chương trình không bị lỗi *RecursionError* trong các trường hợp cây tìm kiếm bị lệch (skewed tree) hoặc khi mở rộng bài toán cho N lớn hơn một chút để kiểm thử.

4 Subtask 2: Phân tích thuật toán

Để đảm bảo thuật toán Metaheuristic hoạt động hiệu quả trong môi trường ngôn ngữ Python (vốn chậm hơn C++), bài giải đã áp dụng các kỹ thuật tối ưu hóa mức thấp (Low-level Optimization) sau:

4.1 Tối ưu cấu trúc dữ liệu (Data Transformation)

Thay vì sử dụng Class Objects để lưu trữ thông tin công việc (việc truy xuất thuộc tính object trong Python tốn chi phí overhead lớn), thuật toán chuyển đổi toàn bộ dữ liệu sang dạng Tuple thuần túy: (id, r, t, p, eff).

- Kỹ thuật này giúp giảm đáng kể thời gian truy xuất bộ nhớ trong vòng lặp chính.
- Thử nghiệm thực tế cho thấy tốc độ cải thiện khoảng $20\% - 30\%$.

4.2 Tiền xử lý dữ liệu (Pre-calculation)

Trong giai đoạn "Tái tạo"(Recreate), thuật toán cần chọn ra các ứng viên tốt nhất. Thay vì sắp xếp lại danh sách ứng viên (tốn $O(N \log N)$) ở mỗi vòng lặp, thuật toán thực hiện sắp xếp trước 3 danh sách riêng biệt ngay từ đầu:

1. sorted_eff: Sắp xếp theo hiệu suất giảm dần.
2. sorted_profit: Sắp xếp theo lợi nhuận giảm dần.
3. sorted_deadline: Sắp xếp theo deadline tăng dần.

Khi cần chọn ứng viên theo chiến lược nào, thuật toán chỉ cần duyệt tuyến tính trên danh sách đã sắp xếp sẵn tương ứng, đưa độ phức tạp của bước chọn ứng viên về $O(N)$.

4.3 Cơ chế Phá hủy và Quản lý Đa dạng (Destruction Mechanism & Diversity Management)

Để tránh việc thuật toán bị kẹt tại các điểm tối ưu cục bộ (Local Optima Risk), chúng tôi thiết kế cơ chế Phá hủy (Ruin) bao gồm hai chiến thuật bổ trợ lẫn nhau. Đây không chỉ là việc xóa ngẫu nhiên, mà là chiến lược quản lý sự đa dạng của nghiệm:

4.3.1 Chiến thuật Phân tán Ngẫu nhiên (Random Scatter)

- **Cơ chế:** Xóa ngẫu nhiên k công việc ($k \in [3, 12]$) từ bất kỳ vị trí nào trong lịch trình.
- **Mục đích - Đa dạng hóa (Diversification):** Trong bài toán Knapsack, đôi khi việc giữ một công việc có lợi nhuận trung bình lại ngăn cản việc chọn hai công việc nhỏ hơn nhưng có tổng lợi nhuận lớn hơn. Random Scatter giúp "rung lắc"tổ hợp các công việc hiện tại, loại bỏ ngẫu nhiên các mắt xích yếu để tạo cơ hội cho các tổ hợp mới xuất hiện.

4.3.2 Chiến thuật Cắt bỏ Theo cụm (Burst Removal)

- **Cơ chế:** Xóa một đoạn liên tiếp k công việc ($k \in [3, 15]$) trên trục thời gian của lịch trình.
- **Mục đích - Sửa lỗi Cấu trúc (Structural Repair):** Trong bài toán Lập lịch, thứ tự thực hiện rất quan trọng. Một công việc có thời gian thực hiện (r_i) lớn nằm sai chỗ (ví dụ: nằm quá sớm) sẽ đẩy lùi thời gian bắt đầu của tất cả công việc phía sau, gây vi phạm deadline hàng loạt.
- *Ví dụ:* Nếu một khối đá lớn chắn ngang đường, việc nhặt sỏi xung quanh (Random Scatter) không giải quyết được vấn đề. Burst Removal giúp "khoét"một khoảng trống thời gian lớn liền mạch, tạo điều kiện để thuật toán sắp xếp lại các khối công việc lớn (Big blocks rearrangement) một cách tối ưu hơn.

4.4 Chiến lược Tái tạo Nghiệm (Recreate Strategy)

Sau khi phá hủy một phần lịch trình, giai đoạn Tái tạo sẽ cố gắng chèn lại các công việc (từ tập các công việc chưa được chọn) để khôi phục một lời giải hoàn chỉnh.

4.4.1 Lựa chọn Chiến lược (Adaptive Selection)

Thuật toán không cố định một cách chèn mà lựa chọn dựa trên xác suất (Roulette Wheel Selection) từ 4 chiến lược heuristic:

1. **Greedy by Efficiency (50%):** Ưu tiên chèn các công việc có mật độ giá trị cao nhất (p_i/r_i). Đây là heuristic mạnh nhất cho bài toán dạng Knapsack.
2. **Greedy by Profit (20%):** Ưu tiên công việc có lợi nhuận p_i tuyệt đối cao nhất (bất chấp thời gian thực hiện dài). Chiến lược này hiệu quả khi quỹ thời gian còn trống nhiều.
3. **Greedy by Deadline (10%):** Ưu tiên công việc có deadline t_i sớm nhất. Chiến lược này giúp tăng tính khả thi (feasibility), đảm bảo các công việc gấp được thực hiện trước.
4. **Balanced Strategy (20%):** Một chiến lược hỗn hợp để cân bằng các yếu tố trên.

4.4.2 Quy trình Chèn (Insertion Process)

Sau khi chọn được ta sẽ tham lam theo tiêu chí nào, bây giờ ta chèn như chèn trong một thuật toán tham lam bình thường thôi:

- **Bước 1: Lọc ứng viên.** Từ danh sách các công việc chưa được chọn, lấy ra các ứng viên tốt nhất theo tiêu chí của chiến lược hiện tại (đã được Pre-sort).
- **Bước 2: Tìm vị trí chèn.** Với mỗi ứng viên, thuật toán quét tuyến tính (hoặc nhị phân) trên lịch trình hiện tại để tìm vị trí chèn sao cho bảo toàn tính sắp xếp theo Deadline (t_i).
- **Bước 3: Kiểm tra tính hợp lệ (Validity Check).** Sau khi chèn thử, tính toán lại tổng thời gian tích lũy. Nếu tại bất kỳ điểm nào $Time_{accumulated} > Deadline$, việc chèn bị hủy bỏ ngay lập tức.

Tối ưu hóa: Việc sử dụng danh sách ứng viên giới hạn ($Top - K$) và kiểm tra hợp lệ ngay lập tức (Inline Validity Check) giúp giai đoạn này duy trì độ phức tạp thấp, cho phép thuật toán thực hiện hàng nghìn vòng lặp trong thời gian giới hạn.

4.5 Cơ chế Học Thích nghi (Adaptive Learning Mechanism)

Đây là thành phần cốt lõi biến LNS (Large Neighborhood Search) thành ALNS. Thay vì gán cứng xác suất cố định cho các chiến thuật tái tạo, thuật toán áp dụng cơ chế học tăng cường (Reinforcement Learning) đơn giản để tự điều chỉnh hành vi theo dữ liệu thực tế:

- **Nguyên lý Vòng quay Roulette (Roulette Wheel Selection):** Xác suất chọn chiến thuật i tại mỗi bước được tính dựa trên trọng số w_i của nó:

$$P(i) = \frac{w_i}{\sum_j w_j}$$

  Điều này đảm bảo các chiến thuật tốt có cơ hội xuất hiện cao hơn, nhưng các chiến thuật ít hiệu quả vẫn có cơ hội nhỏ để được chọn (nhằm duy trì sự đa dạng).

- **Hệ thống Điểm thưởng (Dynamic Reward System):** Ban đầu, các trọng số được khởi tạo dựa trên kinh nghiệm thực nghiệm $[50, 20, 10, 20]$. Tuy nhiên, trong quá trình chạy, nếu chiến thuật i giúp tìm ra một nghiệm mới tốt hơn nghiệm tốt nhất toàn cục (*Global Best*), trọng số của nó lập tức được thưởng thêm một lượng δ:

$$w_i \leftarrow w_i + 5$$

- **Ý nghĩa Tối ưu hóa:** Cơ chế này cho phép thuật toán tự động "nhận diện"đặc trưng của bộ dữ liệu đầu vào (Instance-specific tuning) ngay trong thời gian chạy thực (Runtime). *Ví dụ:* Nếu bộ dữ liệu có deadline rất thoải mái, chiến thuật "Greedy by Deadline"sẽ ít hiệu quả. Thuật toán sẽ dần nhận ra điều này (do chiến thuật đó không sinh ra Global Best) và tự động chuyển hướng tài nguyên sang tập trung vào chiến thuật "Greedy by Profit"hoặc "Efficiency".

5 Phân tích độ phức tạp thời gian và không gian:

5.1 Subtask 1:

- **Độ phức tạp thời gian:**
  - *Lý thuyết:* $O(2^N)$. Bản chất thuật toán là duyệt cây nhị phân, với chiều cao cây là N. Tổng số nút tối đa cần duyệt là $2^{N+1} - 1$.
  - *Thực tế:* Thời gian chạy thực tế là $O(b^N)$ với $b \ll 2$ (hệ số rẽ nhánh hiệu dụng nhỏ hơn 2). Nhờ kỹ thuật **Cắt tỉa (Pruning)** và sắp xếp tiền xử lý ($O(N \log N)$), đa số các nhánh không thỏa mãn điều kiện $current\_time + r_i \leq t_i$ bị loại bỏ rất sớm, giúp thuật toán chạy cực nhanh với $N = 20$.
- **Độ phức tạp không gian:** $O(N)$.
  - Các biến lưu trữ trạng thái trung gian để thực hiện thuật toán (idx, current-time, current-profit, current-path) và các list để lưu dữ liệu nên cũng chỉ chiếm $O(N)$.

5.2 Subtask 2: Giải thuật Metaheuristic (ALNS)

Với Subtask 2, độ phức tạp không phụ thuộc vào kích thước đầu vào theo cách truyền thống, mà phụ thuộc vào giới hạn thời gian cho phép.

- **Độ phức tạp thời gian:**.
  - *Tiền xử lý:* Chi phí sắp xếp 3 danh sách đầu vào là $O(N \log N)$. Đây là chi phí cố định thực hiện 1 lần.
  - *Chi phí trên mỗi vòng lặp (Per-iteration cost):*
    - Giai đoạn Ruin: Thao tác xóa phần tử (‘pop‘) trong mảng Python có độ phức tạp $O(N)$ (do phải dời dịch các phần tử phía sau). Xóa M phần tử tốn $O(M \cdot N)$.
    - Giai đoạn Recreate: Thao tác chèn (‘insert‘) và kiểm tra hợp lệ cũng tốn $O(N)$. Chèn M phần tử tốn $O(M \cdot N)$.

– *Tổng quát:* Tổng thời gian $T(N) \approx K \times O(N^2)$ hay $O(K \times N^2)$, với K là số vòng lặp thực hiện được. Do $T(N)$ bị chặn bởi $T_{limit} = 4.85s$, thuật toán sẽ tự động điều chỉnh K (với N lớn, K sẽ giảm và ngược lại).

- **Độ phức tạp không gian:** $O(N)$.
  - *Lưu trữ dữ liệu:* Cần 3 danh sách phụ trợ đã sắp xếp ('sorted-eff', 'sorted-profit', 'sorted-deadline'), mỗi danh sách kích thước N. Tổng chi phí: $3N$.
  - *Cấu trúc nghiệm:* Lưu trữ lịch trình hiện tại và lịch trình tốt nhất (*best_sched*), mỗi cái kích thước tối đa N.
  - Tổng không gian bộ nhớ duy trì ở mức tuyến tính $O(N)$, hoàn toàn phù hợp với giới hạn bộ nhớ đề bài (thường là 256MB hoặc 512MB) ngay cả khi N lên tới hàng triệu.

6 Code python:

```python
import sys
import time
import random
import math

# Increase recursion depth for Subtask 1
sys.setrecursionlimit(5000)

class Task:
    def __init__(self, id, r, t, p):
        self.id = id
        self.r = r
        self.t = t
        self.p = p

# ------------------------------------------------------------
# SOLVER 1: EXACT BACKTRACKING (N <= 20)
# ------------------------------------------------------------
best_profit_exact = -1
best_path_exact = []

def run_exact_solver(tasks, N):
    tasks.sort(key=lambda x: x.t)
    global best_profit_exact, best_path_exact
    best_profit_exact = -1
    best_path_exact = []

    def backtrack(idx, current_time, current_profit, current_path):
        global best_profit_exact, best_path_exact
        if idx == N:
            if current_profit > best_profit_exact:
                best_profit_exact = current_profit
                best_path_exact = list(current_path)
            return

        task = tasks[idx]
        if current_time + task.r <= task.t:
            current_path.append(task.id)
```

```
            backtrack(idx + 1, current_time + task.r, current_profit + task.p, current_path)
            current_path.pop()

        backtrack(idx + 1, current_time, current_profit, current_path)

    backtrack(0, 0, 0, [])
    print(best_profit_exact)
    print(*(best_path_exact))

# ----------------------------------------------------------
# SOLVER 2: HIGH-PERFORMANCE ALNS (N > 20)
# ----------------------------------------------------------
def solve_large(tasks, start_time):
    # 1. DATA TRANSFORMATION (Speed Hack)
    # Convert objects to simple tuples: (id, r, t, p, eff)
    # We work with these tuples exclusively in the loop for speed.

    # ID=0, R=1, T=2, P=3, EFF=4
    task_tuples = []
    for t in tasks:
        eff = t.p / t.r if t.r > 0 else 0
        task_tuples.append((t.id, t.r, t.t, t.p, eff))

    # Pre-calculate sorting lists to avoid repeated sorting
    # Sorted by Efficiency
    sorted_eff = sorted(task_tuples, key=lambda x: x[4], reverse=True)
    # Sorted by Profit
    sorted_profit = sorted(task_tuples, key=lambda x: x[3], reverse=True)
    # Sorted by Deadline (ascending) - Needed for output validity
    sorted_deadline = sorted(task_tuples, key=lambda x: x[2])

    # Initial Solution: Efficiency Greedy
    current_sched = []
    current_profit = 0

    # Fast Greedy Construction
    for task in sorted_eff:
        # Linear Insert
        # Since we use tuples, accessing task[2] (deadline) is fast
        pos = len(current_sched)
        for i, existing in enumerate(current_sched):
            if task[2] < existing[2]:
                pos = i
                break
        current_sched.insert(pos, task)

        # Check Validity inline
        valid = True
        t_time = 0
        for t in current_sched:
            t_time += t[1] # t.r
            if t_time > t[2]: # t.t
                valid = False
                break

        if valid:
            current_profit += task[3]
```

```
        else:
            current_sched.pop(pos)

    # Global State
    best_profit = current_profit
    best_sched = list(current_sched)

    # Pools for reconstruction
    # We maintain a set of IDs for O(1) lookup
    all_map = {t[0]: t for t in task_tuples}

    iteration = 0

    # Weights for adaptive selection
    # [Efficiency, Profit, Duration, Balanced]
    strat_weights = [50, 20, 10, 20]

    while True:
        iteration += 1
        # Check time less often to save overhead (every 200 iters)
        if iteration % 200 == 0:
            if time.time() - start_time > 4.85:
                break

        # --- RUIN ---
        # 30% chance to restart from best
        if random.random() < 0.3:
            work_sched = list(best_sched)
            work_profit = best_profit
        else:
            work_sched = list(current_sched)
            work_profit = current_profit

        # Removal logic
        # Optimize: Avoid random.choice overhead. Use direct index.
        ln = len(work_sched)
        if ln > 2:
            # 70% Random Scatter, 30% Burst
            if random.random() < 0.7:
                # Random Scatter removal (3 to 12 items)
                cnt = random.randint(3, 12)
                for _ in range(cnt):
                    if not work_sched: break
                    idx = int(random.random() * len(work_sched))
                    rem = work_sched.pop(idx)
                    work_profit -= rem[3]
            else:
                # Burst removal (Slice)
                cnt = random.randint(3, 15)
                start = int(random.random() * (ln - 2))
                end = min(ln, start + cnt)
                # Calculate profit loss efficiently
                for k in range(start, end):
                    work_profit -= work_sched[k][3]
                del work_sched[start:end]

        # --- RECREATE ---
        # Identify current IDs
```

```
        curr_ids = {t[0] for t in work_sched}

        # Strategy Select
        rnd = random.random() * sum(strat_weights)
        strat = 0
        acc = 0
        for i, w in enumerate(strat_weights):
            acc += w
            if rnd <= acc:
                strat = i
                break

        # Candidate Selection
        # Instead of sorting fresh every time (slow), use pre-sorted lists and filter
        # Only take top K valid candidates to speed up

        candidates = []
        limit = 30 # Only try inserting top 30 candidates
        found = 0

        source_list = sorted_eff # Default
        if strat == 1: source_list = sorted_profit
        elif strat == 2: source_list = sorted_deadline # Proxy for duration/earliness

        # Add some randomization to the source list traversal?
        # A full shuffle is too slow.
        # We skip items with probability to simulate randomness.

        for t in source_list:
            if t[0] not in curr_ids:
                # 20% chance to skip good candidate to allow variety
                if random.random() < 0.2: continue

                candidates.append(t)
                found += 1
                if found >= limit: break

        # Insertion Loop
        for cand in candidates:
            # Insert maintaining sorted order (Deadline)
            # Binary search is faster than linear scan for larger lists?
            # N is small (500), linear is fine, but let's optimize.
            # Just scan.

            pos = len(work_sched)
            cand_t = cand[2]

            # Fast scan
            for i, existing in enumerate(work_sched):
                if cand_t < existing[2]:
                    pos = i
                    break

            work_sched.insert(pos, cand)

            # Inline Validity Check
```

```
            valid = True
            acc_time = 0
            for t in work_sched:
                acc_time += t[1]
                if acc_time > t[2]:
                    valid = False
                    break

            if valid:
                work_profit += cand[3]
            else:
                work_sched.pop(pos)

        # --- UPDATE ---
        if work_profit > best_profit:
            best_profit = work_profit
            best_sched = list(work_sched)

            # Reward strategy
            strat_weights[strat] += 5

            current_profit = work_profit
            current_sched = list(work_sched)

        elif work_profit >= current_profit:
            current_profit = work_profit
            current_sched = list(work_sched)

        # Late game catch: if we are close to global best, accept it sometimes
        elif work_profit > best_profit * 0.99 and random.random() < 0.05:
            current_profit = work_profit
            current_sched = list(work_sched)

    print(best_profit)
    print(*(t[0] for t in best_sched))

# ---------------------------------------------------------
# MAIN
# ---------------------------------------------------------
def solve():
    start_time = time.time()
    try:
        input_data = sys.stdin.read().split()
    except Exception: return
    if not input_data: return

    iterator = iter(input_data)
    try:
        N = int(next(iterator))
    except StopIteration: return

    tasks = []
    for i in range(1, N + 1):
        r = int(next(iterator))
        t = int(next(iterator))
        p = int(next(iterator))
        tasks.append(Task(i, r, t, p))
```

```
267
268     if N <= 20:
269         run_exact_solver(tasks, N)
270     else:
271         solve_large(tasks, start_time)
272
273 if __name__ == "__main__":
274     solve()
```