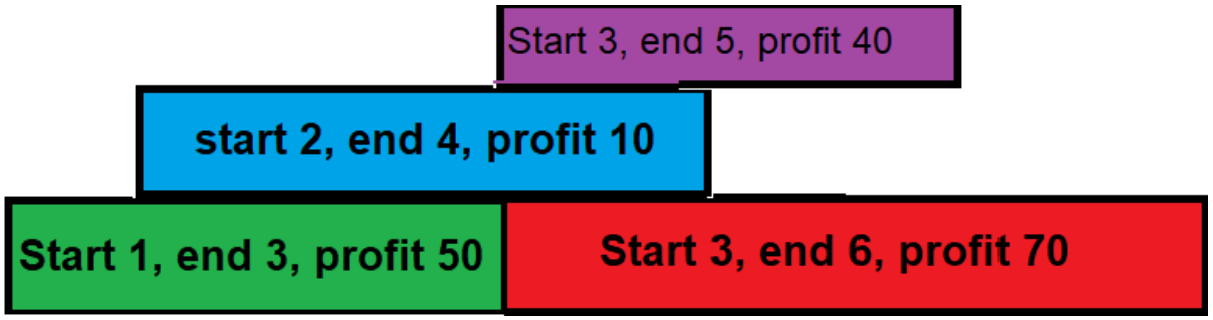
UNIVERSIDADE FEDERAL DE SANTA CATARINA
EDUARDO BISCHOFF GRASEL
FELIPE DE SOUZA GOULART
GUSTAVO GONÇALVES DOS SANTOS

Relatório trabalho prático - SO

Florianópolis
2024

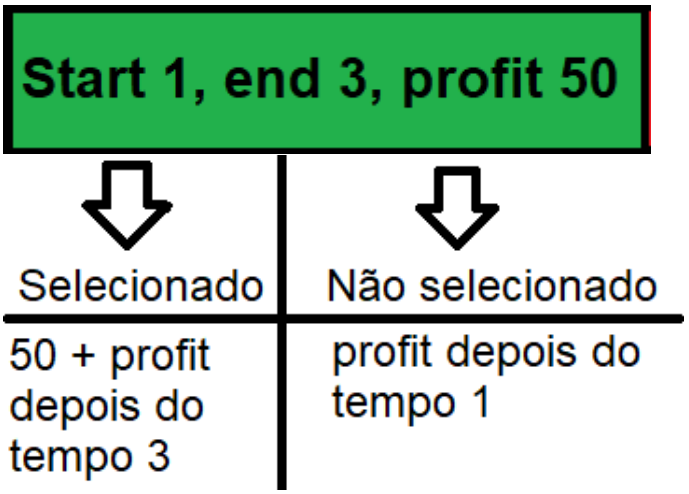
Raciocínio central do algoritmo:

Inicialmente temos que organizar um algoritmo de forma e encontrar o maior profit possível entre jobs, de forma que um não sobreponha o outro:



Nesse caso, escolheríamos o verde e vermelho, pois um não sobrepõe o outro e seu profit é maior do que das outras combinações possíveis, que seria o azul, ou o verde e o roxo.

Isso nos deixa com algumas alternativas para soluções, a mais simples sendo uma força bruta comparando todos com todos, o que não é muito eficiente, entretanto não temos muitas saídas, pois para uma solução perfeita, deveríamos ser capazes de prever o futuro, então devemos otimizar o máximo a lógica de seleção. Imagine que cada um dos jobs tem dois estados possíveis: selecionar ou não selecionar.



Uma vez que um job é verificado devemos fazer uma lógica recursiva de busca e comparações, onde verificamos se percorremos todos os jobs, se existe sobreposição ou um valor já calculado (memorização), uma vez passado por isso, precisamos decidir se incluiremos o job atual ou o próximo, analisando o melhor

profit entre os dois, ao final de todas chamadas recursivas devemos ter a melhor combinação de jobs.

Colocando as ideias em código.

Inicialmente declaramos globalmente um array com 5001 posições para memorização de resultados calculados previamente para otimizar o programa.

Teremos um total de 3 funções, uma para organização dos vetores, outra para a busca de lucro máximo e a que chamará essas duas. Iremos discutir essas funções à medida que são chamadas dentro do código.

Iniciamos com a jobScheduling que recebe os vetores, startTime, endTime, profit. Chamamos o memset para organizar o array de memorização, inicializando todos elementos como -1 para sinalizar que o valor não foi memorizado. Agora juntamos todas as listas e ordenamos pelo startTime.

Agora já temos a organização necessária para iniciar os cálculos, esses serão feitos de maneira recursiva pela função dfs, que é chamada como return do jobScheduling, passando os jobs e a posição inicial (0).

```
int jobScheduling(vector<int> &startTime, vector<int> &endTime, vector<int> &profit) {
    // Inicializa o array 'results' com -1
    memset(results, -1, sizeof results);
    // cria a var para união dos vertores
    vector<vector<int>> jobs;
    // Cria uma matriz para combinar os três vetores
    for (int i = 0; i < startTime.size(); i++) {
        // Cada trabalho é representado como {startTime, endTime, profit}
        jobs.push_back({startTime[i], endTime[i], profit[i]});
    }
    // Ordena os trabalhos pelo tempo de início (startTime)
    sort(jobs.begin(), jobs.end());
    // Chama o DFS para calcular o lucro máximo começando do primeiro trabalho
    return dfs(jobs, 0);
}
```

Precisaremos passar por uma série de condições:

- Caso a posição seja maior do que a quantidade de jobs, significa que não teremos mais um ganho se prosseguirmos;
- Iremos consultar o nosso array de cálculos memorizados, caso o resultado da posição já tenha sido calculado, retornamos ela.

Passando por tudo isso, precisamos decidir se vamos incluir o job atual ou o próximo, para isso vamos considerar os dois casos, iniciando a chamada recursiva para calcular o atual, isso significa que o próximo trabalho que consideramos deverá

começar após o término (endTime) do trabalho atual. O lucro total será o lucro deste trabalho atual (jobs[pos][2]) somado ao resultado da chamada recursiva para os trabalhos subsequentes. Ou ao menos esse era o plano, entretanto ao avaliar o algoritmo se descobriu uma perda de desempenho. Então para aumentar o desempenho, podemos usar uma busca binária para encontrar o próximo trabalho sem superposição para reduzir o número de chamadas recursivas.

```
int nextJobPos = findNextJob(jobs, jobs[pos][1]); // Encontra o próximo trabalho sem sobreposição
int includeCurrent = jobs[pos][2] + dfs(jobs, nextJobPos);
```

Sua implementação é bem simples, apenas dividimos na metade o intervalo de jobs ordenados baseado no endTime atual até que encontremos o índice do próximo trabalho.

```
int findNextJob(vector<vector<int>>& jobs, int currentEndTime) {
    int left = 0, right = jobs.size(); // Inicializa o intervalo de busca
    while (left < right) {             // Enquanto houver intervalo a ser buscado
        int mid = left + (right - left) / 2; // Calcula o meio do intervalo
        if (jobs[mid][0] >= currentEndTime) {
            right = mid;   // Se o trabalho do meio começa depois ou no tempo atual, ajusta o limite superior
        } else {
            left = mid + 1; // Caso contrário, move o limite inferior para o próximo trabalho
        }
    }
    return left; // Retorna o índice do próximo trabalho que pode ser escolhido
}
```

Para pular o trabalho atual chamamos a função recursivamente com o próximo trabalho (pos + 1).

```
int skipCurrent = dfs(jobs, pos + 1);
```

Uma vez que possuímos os dois valores podemos comparar os dois resultados, guardando o maior no nosso array de memorização e retornando.

```
// Retorna o melhor resultado entre incluir ou pular o trabalho atual
return results[pos] = max(includeCurrent, skipCurrent);
```

Conclusão

Inicialmente no algoritmo temos que juntar todos os elementos e ordenar pelo tempo de início, para a junção teremos que percorrer todo array de tempo uma vez, ou seja, complexidade O(n). De acordo com a biblioteca c++ para obtenção do algoritmo sort() temos uma complexidade de O(nlogn). Finalmente a execução da dfs, que percorre os trabalhos com memorização, garantindo que cada trabalho é

processado no máximo uma vez com a ajuda da busca binária, que no pior caso será executada n vezes para cada job: O(nlogn)..

Somando as complexidades temos O(nlogn + n + nlogn), simplificando para O(nlogn).