Chapter 18: Applications of Greedy technique

Section 18.1: Offline Caching

The caching problem arises from the limitation of finite space. Lets assume our cache `C` has `k` pages. Now we want to process a sequence of `m` item requests which must have been placed in the cache before they are processed.Of course if `m<=k` then we just put all elements in the cache and it will work, but usually is `m>>k`.

We say a request is a **cache hit**, when the item is already in cache, otherwise its called a **cache miss**. In that case we must bring the requested item into cache and evict another, assuming the cache is full. The Goal is a eviction schedule that **minimizes the number of evictions**.

There are numerous greedy strategies for this problem, lets look at some:

1. **First in, first out (FIFO)**: The oldest page gets evicted
2. **Last in, first out (LIFO)**: The newest page gets evicted
3. **Last recent out (LRU)**: Evict page whose most recent access was earliest
4. **Least frequently requested(LFU)**: Evict page that was least frequently requested
5. **Longest forward distance (LFD)**: Evict page in the cache that is not requested until farthest in the future.

Attention: For the following examples we evict the page with the smallest index, if more than one page could be evicted.

Example (FIFO)

Let the cache size be `k=3` the initial cache `a,b,c` and the request `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c`:

Request	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
cache 1	a	a	d	d	d	d	a	a	a	d	d	d	f	f	f	c
cache 2	b	b	b	e	e	e	e	c	c	c	e	e	e	b	b	b
cache 3	c	c	c	c	b	b	b	b	f	f	f	a	a	a	e	e
cache miss			x	x	x		x	x	x	x	x	x	x	x	x	x

Thirteen cache misses by sixteen requests does not sound very optimal, lets try the same example with another strategy:

Example (LFD)

Let the cache size be `k=3` the initial cache `a,b,c` and the request `a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c`:

Request	**a**	**a**	**d**	**e**	**b**	**b**	**a**	**c**	**f**	**d**	**e**	**a**	**f**	**b**	**e**	**c**
cache 1	a	a	d	e	e	e	e	e	e	e	e	e	e	e	e	c
cache 2	b	b	b	b	b	b	a	a	a	a	a	a	f	f	f	f
cache 3	c	c	c	c	c	c	c	c	f	d	d	d	d	b	b	b
cache miss			x	x		x		x	x			x	x			x

Eight cache misses is a lot better.

Selftest: Do the example for LIFO, LFU, RFU and look what happend.

The following example programm (written in C++) consists of two parts:

The skeleton is a application, which solves the problem dependent on the chosen greedy strategy:

```
#include <iostream>
#include <memory>

using namespace std;

const int cacheSize     = 3;
const int requestLength = 16;

const char request[]    = {'a','a','d','e','b','b','a','c','f','d','e','a','f','b','e','c'};
char cache[]            = {'a','b','c'};

// for reset
char originalCache[]    = {'a','b','c'};


class Strategy {

public:
    Strategy(std::string name) : strategyName(name) {}
    virtual ~Strategy() = default;

    // calculate which cache place should be used
    virtual int apply(int requestIndex)                                     = 0;

    // updates information the strategy needs
    virtual void update(int cachePlace, int requestIndex, bool cacheMiss)    = 0;

    const std::string strategyName;
};

bool updateCache(int requestIndex, Strategy* strategy)
{
    // calculate where to put request
    int cachePlace = strategy->apply(requestIndex);

    // proof whether its a cache hit or a cache miss
    bool isMiss = request[requestIndex] != cache[cachePlace];

    // update strategy (for example recount distances)
    strategy->update(cachePlace, requestIndex, isMiss);

    // write to cache
    cache[cachePlace] = request[requestIndex];

    return isMiss;
}

int main()
{
    Strategy* selectedStrategy[] = { new FIFO, new LIFO, new LRU, new LFU, new LFD };

    for (int strat=0; strat < 5; ++strat)
    {
        // reset cache
        for (int i=0; i < cacheSize; ++i) cache[i] = originalCache[i];

        cout <<"\nStrategy: " << selectedStrategy[strat]->strategyName << endl;

        cout << "\nCache initial: (";
        for (int i=0; i < cacheSize-1; ++i) cout << cache[i] << ",";
```

```
        cout << cache[cacheSize-1] << ")\n\n";

        cout << "Request\t";
        for (int i=0; i < cacheSize; ++i) cout << "cache " << i << "\t";
        cout << "cache miss" << endl;

        int cntMisses = 0;

        for(int i=0; i<requestLength; ++i)
        {
            bool isMiss = updateCache(i, selectedStrategy[strat]);
            if (isMiss) ++cntMisses;

            cout << "   " << request[i] << "\t";
            for (int l=0; l < cacheSize; ++l) cout << "   " << cache[l] << "\t";
            cout << (isMiss ? "x" : "") << endl;
        }

        cout<< "\nTotal cache misses: " << cntMisses << endl;
    }

    for(int i=0; i<5; ++i) delete selectedStrategy[i];
}
```

The basic idea is simple: for every request I have two calls two my strategy:

1. **apply**: The strategy has to tell the caller which page to use
2. **update**: After the caller uses the place, it tells the strategy whether it was a miss or not. Then the strategy may update its internal data. The strategy **LFU** for example has to update the hit frequency for the cache pages, while the **LFD** strategy has to recalculate the distances for the cache pages.

Now lets look of example implementations for our five strategies:

FIFO

```
class FIFO : public Strategy {
public:
    FIFO() : Strategy("FIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
```

```
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

FIFO just needs the information how long a page is in the cache (and of course only relative to the other pages). So the only thing to do is wait for a miss and then make the pages, which where not evicted older. For our example above the program solution is:

```
Strategy: FIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          d          e          c          x
  b          d          e          b          x
  b          d          e          b
  a          a          e          b          x
  c          a          c          b          x
  f          a          c          f          x
  d          d          c          f          x
  e          d          e          f          x
  a          d          e          a          x
  f          f          e          a          x
  b          f          b          a          x
  e          f          b          e          x
  c          c          b          e          x

Total cache misses: 13
```

Thats exact the solution from above.

LIFO

```
class LIFO : public Strategy {
public:
    LIFO() : Strategy("LIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int newest = 0;
```

```
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] < age[newest])
                newest = i;
        }

        return newest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // nothing changed we don't need to update the ages
        if(!cacheMiss)
            return;

        // all old pages get older, the new one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

The implementation of **LIFO** is more or less the same as by **FIFO** but we evict the youngest not the oldest page. The program results are:

```
Strategy: LIFO

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          d          b          c          x
  e          e          b          c          x
  b          e          b          c
  b          e          b          c
  a          a          b          c          x
  c          a          b          c
  f          f          b          c          x
  d          d          b          c          x
  e          e          b          c          x
  a          a          b          c          x
  f          f          b          c          x
  b          f          b          c
  e          e          b          c          x
  c          e          b          c

Total cache misses: 9
```

LRU

```
class LRU : public Strategy {
public:
    LRU() : Strategy("LRU")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    // here oldest mean not used the longest
    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        // all old pages get older, the used one get 0
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;

            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

In case of **LRU** the strategy is independent from what is at the cache page, its only interest is the last usage. The programm results are:

```
Strategy: LRU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          b          d          e          x
  b          b          d          e
  a          b          a          e          x
  c          b          a          c          x
  f          f          a          c          x
  d          f          d          c          x
  e          f          d          e          x
  a          a          d          e          x
```

```
  f          a          f          e          x
  b          a          f          b          x
  e          e          f          b          x
  c          e          c          b          x

Total cache misses: 13
```

LFU

```
class LFU : public Strategy {
public:
    LFU() : Strategy("LFU")
    {
        for (int i=0; i<cacheSize; ++i) requestFrequency[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int least = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(requestFrequency[i] < requestFrequency[least])
                least = i;
        }

        return least;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(cacheMiss)
            requestFrequency[cachePos] = 1;

        else
            ++requestFrequency[cachePos];
    }

private:

    // how frequently was the page used
    int requestFrequency[cacheSize];
};
```

LFU evicts the page uses least often. So the update strategy is just to count every access. Of course after a miss the count resets. The program results are:

```
Strategy: LFU

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          d          c          x
  e          a          d          e          x
  b          a          b          e          x
  b          a          b          e
  a          a          b          e
```

```
  c          a          b          c          x
  f          a          b          f          x
  d          a          b          d          x
  e          a          b          e          x
  a          a          b          e
  f          a          b          f          x
  b          a          b          f
  e          a          b          e          x
  c          a          b          c          x

Total cache misses: 10
```

LFD

```
class LFD : public Strategy {
public:
    LFD() : Strategy("LFD")
    {
        // precalc next usage before starting to fullfill requests
        for (int i=0; i<cacheSize; ++i) nextUse[i] = calcNextUse(-1, cache[i]);
    }

    int apply(int requestIndex) override
    {
        int latest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(nextUse[i] > nextUse[latest])
                latest = i;
        }

        return latest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
            nextUse[cachePos] = calcNextUse(requestIndex, cache[cachePos]);
    }

private:

    int calcNextUse(int requestPosition, char pageItem)
    {
        for(int i = requestPosition+1; i < requestLength; ++i)
        {
            if (request[i] == pageItem)
                return i;
        }

        return requestLength + 1;
    }

    // next usage of page
    int nextUse[cacheSize];
};
```

The **LFD** strategy is different from everyone before. Its the only strategy that uses the future requests for its decission who to evict. The implementation uses the function `calcNextUse` to get the page which next use is

farthest away in the future. The program solution is equal to the solution by hand from above:

```
Strategy: LFD

Cache initial: (a,b,c)

Request    cache 0    cache 1    cache 2    cache miss
  a          a          b          c
  a          a          b          c
  d          a          b          d          x
  e          a          b          e          x
  b          a          b          e
  b          a          b          e
  a          a          b          e
  c          a          c          e          x
  f          a          f          e          x
  d          a          d          e          x
  e          a          d          e
  a          a          d          e
  f          f          d          e          x
  b          b          d          e          x
  e          b          d          e
  c          c          d          e          x

Total cache misses: 8
```

The greedy strategy **LFD** is indeed the only optimal strategy of the five presented. The proof is rather long and can be found here or in the book by Jon Kleinberg and Eva Tardos (see sources in remarks down below).

Algorithm vs Reality

The **LFD** strategy is optimal, but there is a big problem. Its an optimal **offline** solution. In praxis caching is usually an **online** problem, that means the strategy is useless because we cannot now the next time we need a particular item. The other four strategies are also **online** strategies. For online problems we need a general different approach.

Section 18.2: Ticket automat

First simple Example:

You have a ticket automat which gives exchange in coins with values 1, 2, 5, 10 and 20. The dispension of the exchange can be seen as a series of coin drops until the right value is dispensed. We say a dispension is **optimal** when its **coin count is minimal** for its value.

Let `M` in `[1,50]` be the price for the ticket `T` and `P` in `[1,50]` the money somebody paid for `T`, with `P >= M`. Let `D=P-M`. We define the **benefit** of a step as the difference between `D` and `D-c` with `c` the coin the automat dispense in this step.

The **Greedy Technique** for the exchange is the following pseudo algorithmic approach:

Step 1: while `D > 20` dispense a 20 coin and set `D = D - 20`
Step 2: while `D > 10` dispense a 10 coin and set `D = D - 10`
Step 3: while `D > 5` dispense a 5 coin and set `D = D - 5`
Step 4: while `D > 2` dispense a 2 coin and set `D = D - 2`
Step 5: while `D > 1` dispense a 1 coin and set `D = D - 1`

Afterwards the sum of all coins clearly equals `D`. Its a **greedy algorithm** because after each step and after each repitition of a step the benefit is maximized. We cannot dispense another coin with a higher benefit.

Now the ticket automat as program (in C++):

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// read some coin values, sort them descending,
// purge copies and guaratee the 1 coin is in it
std::vector<unsigned int> readInCoinValues();

int main()
{
    std::vector<unsigned int> coinValues;   // Array of coin values ascending
    int ticketPrice;                        // M in example
    int paidMoney;                          // P in example

    // generate coin values
    coinValues = readInCoinValues();

    cout << "ticket price: ";
    cin >> ticketPrice;

    cout << "money paid: ";
    cin >> paidMoney;

    if(paidMoney <= ticketPrice)
    {
        cout << "No exchange money" << endl;
        return 1;
    }

    int diffValue = paidMoney - ticketPrice;

    // Here starts greedy

    // we save how many coins we have to give out
    std::vector<unsigned int> coinCount;

    for(auto coinValue  = coinValues.begin();
             coinValue != coinValues.end(); ++coinValue)
    {
        int countCoins = 0;

        while (diffValue >= *coinValue)
        {
            diffValue -= *coinValue;
            countCoins++;
        }

        coinCount.push_back(countCoins);
    }

    // print out result
    cout << "the difference " << paidMoney - ticketPrice
         << " is paid with: " << endl;
```

```
    for(unsigned int i=0; i < coinValues.size(); ++i)
    {
        if(coinCount[i] > 0)
            cout << coinCount[i] << " coins with value "
                 << coinValues[i] << endl;
    }

    return 0;
}

std::vector<unsigned int> readInCoinValues()
{
    // coin values
    std::vector<unsigned int> coinValues;

    // make sure 1 is in vectore
    coinValues.push_back(1);

    // read in coin values (attention: error handling is omitted)
    while(true)
    {
        int coinValue;

        cout << "Coin value (<1 to stop): ";
        cin >> coinValue;

        if(coinValue > 0)
            coinValues.push_back(coinValue);

        else
            break;
    }

    // sort values
    sort(coinValues.begin(), coinValues.end(), std::greater<int>());

    // erase copies of same value
    auto last = std::unique(coinValues.begin(), coinValues.end());
    coinValues.erase(last, coinValues.end());

    // print array
    cout << "Coin values: ";

    for(auto i : coinValues)
        cout << i << " ";

    cout << endl;

    return coinValues;
}
```

Be aware there is now input checking to keep the example simple. One example output:

```
Coin value (<1 to stop): 2
Coin value (<1 to stop): 4
Coin value (<1 to stop): 7
Coin value (<1 to stop): 9
Coin value (<1 to stop): 14
Coin value (<1 to stop): 4
Coin value (<1 to stop): 0
```

```
Coin values: 14 9 7 4 2 1
ticket price: 34
money paid: 67
the difference 33 is paid with:
2 coins with value 14
1 coins with value 4
1 coins with value 1
```

As long as `1` is in the coin values we now, that the algorithm will terminate, because:

- `D` strictly decreases with every step
- `D` is never `>0` and smaller than than the smallest coin `1` at the same time

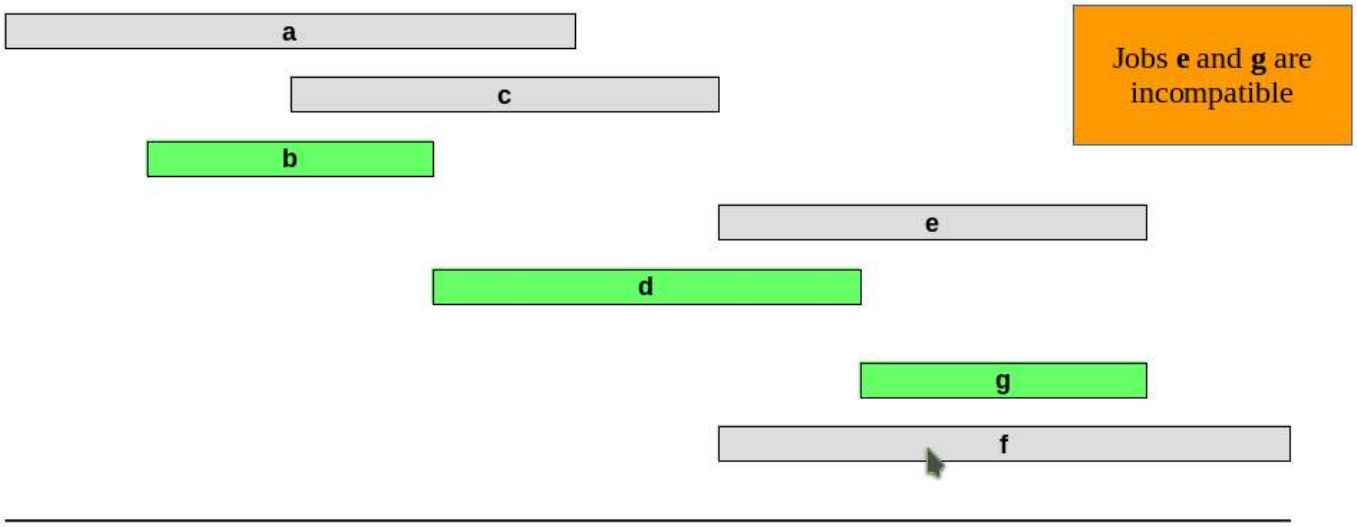
But the algorithm has two pitfalls:

1. Let `C` be the biggest coin value. The runtime is only polynomial as long as `D/C` is polynomial, because the representation of `D` uses only `log D` bits and the runtime is at least linear in `D/C`.
2. In every step our algorithm chooses the local optimum. But this is not sufficient to say that the algorithm finds the global optimal solution (see more information here or in the Book of Korte and Vygen).

A simple counter example: the coins are `1,3,4` and `D=6`. The optimal solution is clearly two coins of value `3` but greedy chooses `4` in the first step so it has to choose `1` in step two and three. So it gives no optimal soution. A possible optimal Algorithm for this example is based on **dynamic programming**.

Section 18.3: Interval Scheduling

We have a set of jobs `J={a,b,c,d,e,f,g}`. Let `j in J` be a job than its start at `sj` and ends at `fj`. Two jobs are compatible if they don't overlap. A picture as example:



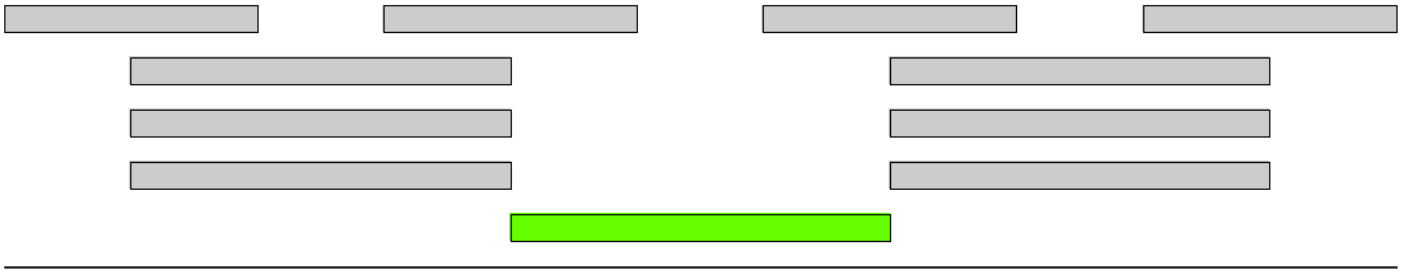
The goal is to find the **maximum subset of mutually compatible jobs**. There are several greedy approaches for this problem:

1. **Earliest start time**: Consider jobs in ascending order of `sj`
2. **Earliest finish time**: Consider jobs in ascending order of `fj`
3. **Shortest interval**: Consider jobs in ascending order of `fj-sj`
4. **Fewest conflicts**: For each job `j`, count the number of conflicting jobs `cj`

The question now is, which approach is really successfull. **Early start time** definetly not, here is a counter example

Shortest interval is not optimal either

and **fewest conflicts** may indeed sound optimal, but here is a problem case for this approach:



Which leaves us with **earliest finish time**. The pseudo code is quiet simple:

1. Sort jobs by finish time so that `f1<=f2<=...<=fn`
2. Let `A` be an empty set
3. for `j=1` to `n` if `j` is compatible to **all** jobs in `A` set `A=A+{j}`
4. `A` is a **maximum subset of mutually compatible jobs**

Or as C++ program:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int startTimes[] = { 2, 3, 1, 4, 3, 2, 6, 7, 8, 9};

// Job end times
const int endTimes[]   = { 4, 4, 3, 5, 5, 5, 8, 9, 9, 10};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(startTimes[i], endTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                    { return p1.second < p2.second; });
```

```
    // step 2: empty set A
    vector<int> A;

    // step 3:
    for(int i=0; i<jobCnt; ++i)
    {
        auto job = jobs[i];
        bool isCompatible = true;

        for(auto jobIndex : A)
        {
            // test whether the actual job and the job from A are incompatible
            if(job.second >= jobs[jobIndex].first &&
               job.first  <= jobs[jobIndex].second)
            {
                isCompatible = false;
                break;
            }
        }

        if(isCompatible)
            A.push_back(i);
    }

    //step 4: print A
    cout << "Compatible: ";

    for(auto i : A)
        cout << "(" << jobs[i].first << "," << jobs[i].second << ") ";
    cout << endl;

    return 0;
}
```

The output for this example is: `Compatible: (1,3) (4,5) (6,8) (9,10)`

The implementation of the algorithm is clearly in Θ(n^2). There is a Θ(n log n) implementation and the interested reader may continue reading below (Java Example).

Now we have a greedy algorithm for the interval scheduling problem, but is it optimal?

Proposition: The greedy algorithm **earliest finish time** is optimal.

Proof:*(by contradiction)*

Assume greedy is not optimal and `i1,i2,...,ik` denote the set of jobs selected by greedy. Let `j1,j2,...,jm` denote the set of jobs in an **optimal** solution with `i1=j1,i2=j2,...,ir=jr` for the **largest possible** value of `r`.

The job `i(r+1)` exists and finishes before `j(r+1)` (earliest finish). But than is `j1,j2,...,jr,i(r+1),j(r+2),...,jm` also a **optimal** solution and for all `k` in `[1,(r+1)]` is `jk=ik`. thats a **contradiction** to the maximality of `r`. This concludes the proof.

This second example demonstrates that there are usually many possible greedy strategies but only some or even none might find the optimal solution in every instance.

Below is a Java program that runs in Θ(n log n)

```
import java.util.Arrays;
```

```
import java.util.Comparator;

class Job
{
    int start, finish, profit;

    Job(int start, int finish, int profit)
    {
        this.start = start;
        this.finish = finish;
        this.profit = profit;
    }
}


class JobComparator implements Comparator<Job>
{
    public int compare(Job a, Job b)
    {
        return a.finish < b.finish ? -1 : a.finish == b.finish ? 0 : 1;
    }
}

public class WeightedIntervalScheduling
{
    static public int binarySearch(Job jobs[], int index)
    {
        int lo = 0, hi = index - 1;

        while (lo <= hi)
        {
            int mid = (lo + hi) / 2;
            if (jobs[mid].finish <= jobs[index].start)
            {
                if (jobs[mid + 1].finish <= jobs[index].start)
                    lo = mid + 1;
                else
                    return mid;
            }
            else
                hi = mid - 1;
        }

        return -1;
    }

    static public int schedule(Job jobs[])
    {
        Arrays.sort(jobs, new JobComparator());

        int n = jobs.length;
        int table[] = new int[n];
        table[0] = jobs[0].profit;

        for (int i=1; i<n; i++)
        {
            int inclProf = jobs[i].profit;
            int l = binarySearch(jobs, i);
            if (l != -1)
                inclProf += table[l];

            table[i] = Math.max(inclProf, table[i-1]);
```

```
        }

        return table[n-1];
    }

    public static void main(String[] args)
    {
        Job jobs[] = {new Job(1, 2, 50), new Job(3, 5, 20),
                      new Job(6, 19, 100), new Job(2, 100, 200)};

        System.out.println("Optimal profit is " + schedule(jobs));
    }
}
```

And the expected output is:

```
Optimal profit is 250
```

Section 18.4: Minimizing Lateness

There are numerous problems minimizing lateness, here we have a single resource which can only process one job at a time. Job `j` requires `tj` units of processing time and is due at time `dj`. if `j` starts at time `sj` it will finish at time `fj=sj+tj`. We define lateness `L=max{0,fj-dh}` for all `j`. The goal is to minimize the **maximum lateness** L.

	1	**2**	**3**	**4**	**5**	**6**
tj	3	2	1	4	3	2
dj	6	8	9	9	10	11

Job	**3**	**2**	**2**	**5**	**5**	**5**	**4**	**4**	**4**	**4**	**1**	**1**	**1**	**6**	**6**
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Lj	-8		-5			-4				1			**7**		4

The solution `L=7` is obviously not optimal. Lets look at some greedy strategies:

1. **Shortest processing time first**: schedule jobs in ascending order og processing time j`
2. **Earliest deadline first**: Schedule jobs in ascending order of deadline `dj`
3. **Smallest slack**: schedule jobs in ascending order of slack `dj-tj`

Its easy to see that **shortest processing time first** is not optimal a good counter example is

	1	**2**
tj	1	5
dj	10	5

the **smallest stack** solution has simillar problems

	1	**2**
tj	1	5
dj	3	5

the last strategy looks valid so we start with some pseudo code:

1. Sort `n` jobs by due time so that `d1<=d2<=...<=dn`
2. Set `t=0`
3. for `j=1` to `n`
    - Assign job `j` to interval `[t,t+tj]`

- set `sj=t` and `fj=t+tj`
- set `t=t+tj`

4. return intervals `[s1,f1],[s2,f2],...,[sn,fn]`

And as implementation in C++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>

const int jobCnt = 10;

// Job start times
const int processTimes[] = { 2, 3, 1, 4, 3, 2, 3, 5, 2, 1};

// Job end times
const int dueTimes[]     = { 4, 7, 9, 13, 8, 17, 9, 11, 22, 25};

using namespace std;

int main()
{
    vector<pair<int,int>> jobs;

    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(processTimes[i], dueTimes[i]));

    // step 1: sort
    sort(jobs.begin(), jobs.end(),[](pair<int,int> p1, pair<int,int> p2)
                                  { return p1.second < p2.second; });

    // step 2: set t=0
    int t = 0;

    // step 3:
    vector<pair<int,int>> jobIntervals;

    for(int i=0; i<jobCnt; ++i)
    {
        jobIntervals.push_back(make_pair(t,t+jobs[i].first));
        t += jobs[i].first;
    }

    //step 4: print intervals
    cout << "Intervals:\n" << endl;

    int lateness = 0;

    for(int i=0; i<jobCnt; ++i)
    {
        auto pair = jobIntervals[i];

        lateness = max(lateness, pair.second-jobs[i].second);

        cout << "(" << pair.first << "," << pair.second << ") "
             << "Lateness: " << pair.second-jobs[i].second << std::endl;
    }

    cout << "\nmaximal lateness is " << lateness << endl;
```

```
    return 0;
}
```

And the output for this program is:

```
Intervals:

(0,2)   Lateness:-2
(2,5)   Lateness:-2
(5,8)   Lateness: 0
(8,9)   Lateness: 0
(9,12)  Lateness: 3
(12,17) Lateness: 6
(17,21) Lateness: 8
(21,23) Lateness: 6
(23,25) Lateness: 3
(25,26) Lateness: 1

maximal lateness is 8
```

The runtime of the algorithm is obviously Θ(n log n) because sorting is the dominating operation of this algorithm. Now we need to show that it is optimal. Clearly an optimal schedule has no **idle time**. the **earliest deadline first** schedule has also no idle time.

Lets assume the jobs are numbered so that `d1<=d2<=...<=dn`. We say a **inversion** of a schedule is a pair of jobs `i` and `j` so that `i<j` but j is scheduled before `i`. Due to its definition the **earliest deadline first** schedule has no inversions. Of course if a schedule has an inversion it has one with a pair of inverted jobs scheduled consecutively.

Proposition: Swapping two adjacent, inverted jobs reduces the number of inversions by **one** and **does not increase** the maximal lateness.

Proof: Let `L` be the lateness before the swap and `M` the lateness afterwards. Because exchanging two adjacent jobs does not move the other jobs from their position it is `Lk=Mk` for all `k != i,j`.

Clearly it is `Mi<=Li` since job `i` got scheduled earlier. if job `j` is late, so follows from the definition:

```
Mj = fi-dj    (definition)
  <= fi-di    (since i and j are exchanged)
  <= Li
```

That means the lateness after swap is less or equal than before. This concludes the proof.

Proposition: The **earliest deadline first** schedule `S` is optimal.

Proof:*(by contradiction)*

Lets assume `S*` is optimal schedule with the **fewest possible** number of inversions. we can assume that `S*` has no idle time. If `S*` has no inversions, then `S=S*` and we are done. If `S*` has an inversion, than it has an adjacent inversion. The last Proposition states that we can swap the adjacent inversion without increasing lateness but with decreasing the number of inversions. This contradicts the definition of `S*`.

The minimizing lateness problem and its near related **minimum makespan** problem, where the question for a minimal schedule is asked have lots of applications in the real world. But usually you don't have only one machine but many and they handle the same task at different rates. These problems get NP-complete really fast.

Another interesting question arises if we don't look at the **offline** problem, where we have all tasks and data at hand but at the **online** variant, where tasks appear during execution.