Applied Algorithm Design
Lecture 3

Pietro Michiardi

Eurecom

PART I : GREEDY ALGORITHMS

Greedy algorithms

It is very hard to define precisely what is meant by a *greedy algorithm*.

- An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.
- When a greedy algorithm arrives at an optimal solution for a non trivial problem, this typically implies something interesting and useful on the structure of the problem: there is a local decision rule that one can use to construct optimal solutions!
- Note: it is easy to invent greedy algorithms for almost any problem; however, finding cases in which they work well and proving that they work well is the interesting challenge.

Content of part I

- Develop two basic methods for proving that a greedy strategy is optimal
  - **Greedy stays ahead:** here we measure the progress of the greedy algorithm and show, in a step-by-step way, that it does better than any other algorithm.
  - **Exchange argument:** here we consider any possible solution to the problem and gradually transform it to the solution found by the greedy algorithm without hurting its quality.
- Apply these concepts to analyze some practical problems in which greedy algorithms can be used

Interval scheduling problem

Interval scheduling

We have a set of requests $\{1, ..., n\}$ where the i^{th} request corresponds to an interval of time starting at $s(i)$ and finishing at $f(i)$.

We say that a subset of the requests is *compatible* if no two of them overlap in time.

Our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called *optimal*.

Interval scheduling: the greedy algorithm stays ahead

We now design a greedy algorithm:

- The basic idea here is to use a simple rule to select a first request i_1;
- Once a request i_1 is accepted, we reject all requests that are not compatible with i_1;
- We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2;
- We continue this way until we run out of requests

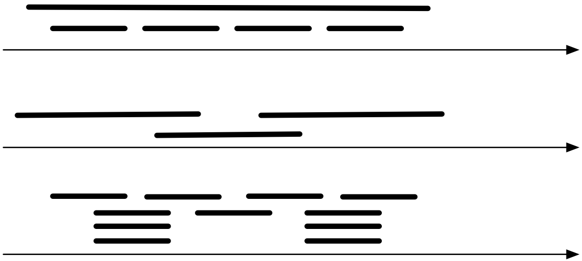
The problem:

How do we decide a simple rule to use for the selection of intervals?

Interval scheduling: attempt 1

- Always select the available request that starts earliest, that is the one with the minimum $s(i)$.
- The idea here is to select intervals so that our resource starts being used as quickly as possible

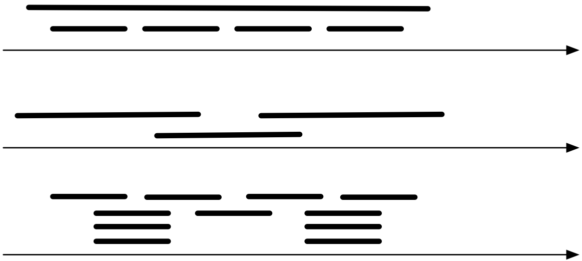
Does it work?



Interval scheduling: attempt 2

- Select the request that requires the smallest interval of time, that is the interval for which $f(i) - s(i)$ is as small as possible

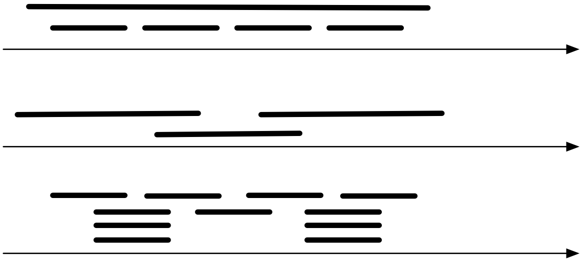
Does it work?



Interval scheduling: attempt 3

- For each request, count the number of other requests that are not compatible and accept the request that has the fewest number of "conflicts".

Does it work?



Interval scheduling: attempt 4

- Accept the request that finishes first, that is the request for which $f(i)$ is as small as possible.
- This is actually a natural approach: we ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests.

Interval scheduling: Algorithm based on the last idea

```
Let $R$ be the set of all requests.
Let $A$ be the set of accepted requests, which is initially empty.
while $R$ is not empty do
    Chose $i \in R$ that has the smallest finishing time
    Add request $i$ to $A$
    Delete all requests from $R$ that are not compatible with $i$
end
Return the set $A$ as the set of accepted requests.
```

Interval scheduling: Algorithm Analysis (1)

- The first thing we can say about the algorithm is that the set A contains compatible requests

Proposition:

A is a compatible set of requests

What we need to show is that this solution is optimal.

Interval scheduling: Algorithm Analysis (2)

- Let $\mathcal{O}$ be an optimal set of intervals.
- Ideally, one may want to show that $A = \mathcal{O}$.
- However, there may be many optimal solutions, and at best A will be equal to one of them.
- Hence, we will show that $|A| = |\mathcal{O}|$, that is A contains the same number of intervals as the optimal solution.

Interval scheduling: Algorithm Analysis (3)

We want to show that our *greedy algorithm stays ahead*.

- We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution $\mathcal{O}$
- Then we show that the algorithm is somehow doing better, step-by-step.

First, let's introduce some notation:

- Let $i_1, ..., i_k$ be the set of requests in A in the order they were added to A.
- Note that $|A| = k$
- Similarly, let $j_1, ..., j_m$ be the set of requests in $\mathcal{O}$.
- We want to show that $k = m$

Interval scheduling: Algorithm Analysis (4)

Assume that the requests in $\mathcal{O}$ are also ordered in the natural left-to-right order of the corresponding intervals, that is in the order of the start and finish points.

Note:

The requests in $\mathcal{O}$ are compatible, which implies the start points have the same order as the finish points.

Our intuition for the last attempt came from wanting our resource to become free again as soon as possible after satisfying the first request.

Interval scheduling: Algorithm Analysis (5)

Observation:

Our greedy rule guarantees that $f(i_1) \leq f(j_1)$.
This is the sense in which we want to prove that our algorithm “stays ahead”: each of the intervals in A finishes at least as soon as the corresponding interval in $\mathcal{O}$.

We now prove that for each $r \geq 1$, the r^{th} accepted request in the algorithm’s schedule finishes no later than the r^{th} in the optimal schedule.

Interval scheduling: Algorithm Analysis (6)

Lemma:

For all indexes $r \leq k$ we have that $f(i_r) \leq f(j_r)$

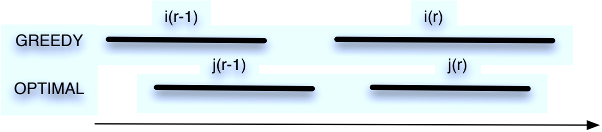
Proof.

By induction.

- $r = 1$ the statement is true: we start by selecting the request i_1 with the minimum finish time.
- Let $r > 1$; the *induction hypothesis* says that the statement is true for $r - 1$, and we will prove it for r.
- As shown in the following figure, the induction hypothesis lets us assume that $f(i_{r-1}) \leq f(j_{r-1})$.
- For the r^{th} interval **not** to finish earlier as well, it would need to “fall behind” as shown.

□

Interval scheduling: Algorithm Analysis (7)



Interval scheduling: Algorithm Analysis (8)

Proof.

- But this cannot happen: rather than choosing a later-finishing interval, the greedy algorithm always has the option (at worst) of choosing j_r and thus fulfilling the induction step.
- We know that: $f(j_{r-1}) \leq s(j_r)$
- Continuing with the induction $f(i_{r-1}) \leq f(j_{r-1})$ we get $f(i_{r-1}) \leq s(j_r)$
- Thus, the interval j_r is in the set R of available intervals when the greedy algorithm selects i_r.
- The greedy algorithm selects the interval with the *smallest* finish time;
- since interval j_r is one of these available intervals, we have $f(i_r) \leq f(j_r)$

□

Interval scheduling: Algorithm Analysis (9)

Theorem:

The greedy algorithm returns an optimal set A.

Proof.

By contradiction.

- If A is not optimal then $\mathcal{O}$ have more requests: $m > k$.
- By the previous Lemma, with $r = k$ we get $f(i_k) \leq f(j_k)$
- Since $m > k$ it means there is a request (in $\mathcal{O}$) that our algorithm didn't explore.
- But this contradicts the stop condition of the algorithm.

□

Interval scheduling: Implementation and running time (1)

```
Sort intervals by finish time: $f_1 \leq f_2 \leq ... \leq f_n$
Initialize set $A = \emptyset$
for $j = 1$ to $n$ do
    if $r_j$ compatible with $A$ then
        $A \leftarrow A \cup \{r_j\}$
    end
end
return $A$
```

Interval scheduling: Implementation and running time (2)

Running time:

Here the most costly operation is to sort:
$O(n\,logn)$ is the running time

Note:

Up to now we assumed inputs (requests) to be available all at once.

What if we receive requests serially? → On-line algorithms.

Interval partitioning problem

Interval partitioning:

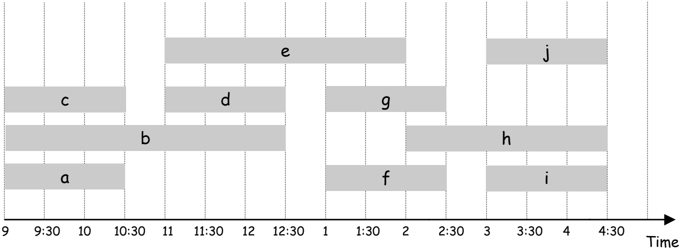
There are many identical resources available and we wish to schedule *all requests* using as few resources as possible.

Here the problem is to partition all intervals across multiple resources.

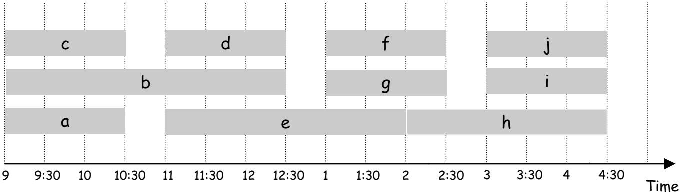
Example:

Suppose that each request corresponds to a lecture that needs to be scheduled in a classroom for a particular interval of time.
We wish to satisfy all these requests, using as few classrooms as possible with the constraint that any two lectures that overlap in time must be scheduled in different classrooms.

Interval partitioning problem: an illustration (1)



Interval partitioning problem: an illustration (2)



Interval partitioning problem: definitions

Depth:

We define the *depth* of a set of intervals to be the maximum number of intervals that pass over any single point on the time-line.

Lemma:

In any instance of the Interval Partitioning Problem, the number of resources needed is at least the depth of the set of intervals.

Proof.

- Suppose a set of intervals has depth d
- Let $I_1, \ldots, I_d$ pass over a common point in time
- Each of these intervals must be scheduled on different resources, so the whole schedule needs at least d resources

□

Interval partitioning problem: towards a greedy algorithm (1)

- Can we design an efficient algorithm that schedules all intervals using the minimum possible number of resources?
- Is there always a schedule using a number of resources that is *equal* to the depth?

A positive answer to the last question would imply that the only obstacle to partitioning intervals is purely local, a set of intervals all piled over the same point. However, it's not immediate that there can't be other obstacles in the "long range".

Interval partitioning problem: designing the algorithm

- Let d be the depth of the set of intervals
- We assign a *label* to each interval, where the labels come from the set $\{1, ..., d\}$
- The assignment has the property that overlapping intervals are labeled with different numbers
- This gives the desired solution, since we can interpret each number as the name of a resource to which an interval is assigned

Interval partitioning problem: designing the algorithm

```
Sort the intervals by their start times, breaking ties arbitrarily
Let $I_1, I_2, ..., I_n$ denote the intervals in this order
for $j = 1, 2, ..., n$ do
    foreach $I_i$ that precedes $I_j$ in sorted order and overlaps it do
        Exclude the label of $I_i$ from consideration for $I_j$
    end
    if There is any label in $\{1, ..., d\}$ that has not been excluded then
        Assign a non-excluded label to $I_j$
    else
        Leave $I_j$ unlabeled
    end
end
```

Interval partitioning problem: analyzing the algorithm

Proposition:

If we use the greedy algorithm above, every interval will be assigned a label, and no two overlapping intervals will receive the same label.

Proof.

- Focus on interval I_j, suppose there are t earlier intervals in the sorted order that overlap it
- We have a set of $t + 1$ intervals that all cross at the start time of I_j, hence $t + 1 \leq d$
- Since $t \leq d - 1$, at least one of the d labels is not excluded
- $\rightarrow$ no interval ends up un-labeled
- Now, focus on intervals I and I' that overlap, and suppose I comes before I'
- When I' is considered by the algorithm, I is in the set of intervals whose labels are excluded, hence I and I' will have different labels
- $\rightarrow$ no two overlapping intervals will have the same label

□

Interval partitioning problem: analyzing the algorithm

Theorem:

The greedy algorithm above schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.

Proof.

We can use the previous lemma, to say that we are always using the minimum number of labels. □

Scheduling to minimize lateness: introduction

Informal problem statement:

We have a single resource and a set of n requests to use the resource for an interval of time. Assume the resource is available at time s. In this problem, each request is more flexible: instead of a start time and a finish time, the request i has a deadline d_i, and requires a **contiguous** time interval of length t_i, but is willing to be scheduled any time before the deadline d_i. Each accepted request must be assigned an interval of time t_i and different requests must be assigned non-overlapping intervals.

Objective:

Can you tell what is an interesting objective function one could seek to optimize here?

Scheduling to minimize lateness: defining our goal

Here we consider a very natural goal that can be optimized by a greedy algorithm.

- Suppose we plan to satisfy each request, but we are allowed to let certain requests run late.
- Our overall start time is s (time origin)
- We will assign each request i an interval of time of length t_i
- Denote this interval by $[s(i), f(i)]$, with $f(i) = s(i) + t_i$
- Unlike previously discussed, our problem is to fix a start time (and hence a finish time)

Scheduling to minimize lateness: some definitions

- We say that request i is **late** if it misses the deadline, that is if $f(i) > d_i$
- The **lateness** of such a request i is $l_i = f(i) - d_i$
- We say that request i is no late if $l_i = 0$
- In the following we will refer to our requests as **jobs**

The problem:

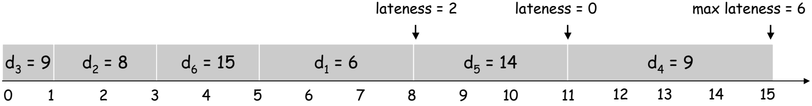
The goal in our optimization problem is to schedule all requests, using **non-overlapping** intervals, so as to minimize the *maximum lateness*, defined as $L = max_i l_i$.

This is a **minimax** problem!

Scheduling to minimize lateness: illustration

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Scheduling to minimize lateness: how can we proceed?

Let's look at some "greedy templates":

- **Shortest processing time first**: consider jobs in ascending order of processing time t_j
- **Smallest slack**: Consider jobs in ascending order of slack $d_j - t_j$
- **Earliest deadline first**: consider jobs in ascending order of deadline d_j (without considering the processing time)

Scheduling to minimize lateness: Shortest processing time first

Idea:

Consider jobs in ascending order of processing time t_j (without considering deadlines)

	1	2
t_j	1	10
d_j	100	10

counterexample

Scheduling to minimize lateness: Smallest slack

Idea:

Consider jobs in ascending order of slack $d_j - t_j$

	1	2
t_j	1	10
d_j	2	10

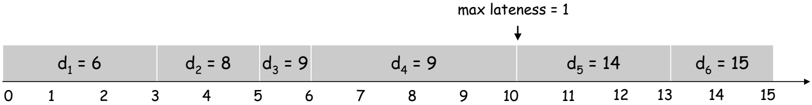
counterexample

Scheduling to minimize lateness: greedy algorithm

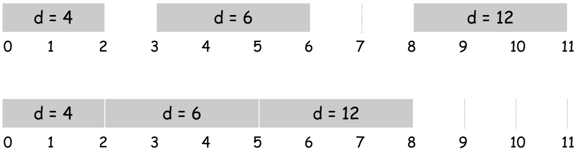
```
Sort $n$ jobs by deadline so that $d_1 \leq d_3 \leq ... \leq d_n$
$t \leftarrow 0$
for $j = 1..n$ do
    Assign job $j$ to interval $[t, t + t_j]$:
    $s_j \leftarrow t$
    $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
end
Output: intervals $[s_j, f_j]$
```

Scheduling to minimize lateness: greedy algorithm

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Scheduling to minimize lateness: some observations



Observation 1:

There exists an optimal schedule with no **idle-time**.

Observation 2:

The greedy algorithm above has no idle time.

Scheduling to minimize lateness: analyzing the algorithm

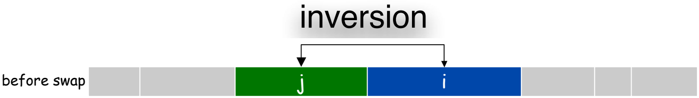
How can we prove that the schedule A defined by the algorithm is optimal, that is, its maximum lateness L is as small as possible?

- Start considering an optimal schedule $\mathcal{O}$
- Gradually modify $\mathcal{O}$ to make it identical to A, while preserving its optimality at each step
- That is, we use the **exchange argument**

Scheduling to minimize lateness: inversions (1)

Definition:

A schedule A' has an *inversion* if a job i with deadline d_i is scheduled before another job j with an earlier deadline $d_j < d_i$.



Observation:

By definition of the algorithm, the schedule A produced by its execution has no inversions.

Scheduling to minimize lateness: inversions (2)

Lemma:

All schedules with no inversions and no idle time have the same maximum lateness.

Proof.

- If two schedules have no inversions nor idle time, they might not produce exactly the same order of jobs
- But, they can only differ in the order in which jobs with identical deadlines are scheduled
- Consider such a deadline *d*
- In both schedules, the jobs with deadline *d* are all scheduled consecutively
- Among the jobs with deadline *d*, the last one has the greatest lateness, and this lateness does not depend on the order of jobs

□

Scheduling to minimize lateness: inversions (3)

Lemma:

There is an optimal schedule that has no inversions and no idle time

Proof.

- By Observation 1, there is an optimal schedule $\mathcal{O}$ with no idle time
- Part A:
  If $\mathcal{O}$ has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$
- We can decrease the number of inversions in $\mathcal{O}$ by swapping i and j, without creating new inversions.

□

Scheduling to minimize lateness: inversions (4)

Proof.

- Part B:
  After swapping i and j we get a schedule with one less inversion
- Part C:
  The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$

□

The initial schedule $\mathcal{O}$ can have at most $\binom{n}{2}$ inversions, that is all pairs are inverted. Hence after $\binom{n}{2}$ swaps we get an optimal schedule without inversions.

Scheduling to minimize lateness: inversions (4)

If we can prove part C, we're done.

- Let l be the lateness before the swap, and let l' be it afterwards
- $l'_k = l_k \forall k \neq i, j$
- $l'_j \leq l_i$
  Indeed: $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i = l_i$

Scheduling to minimize lateness: greedy is optimal

Theorem:

The schedule A produced by the greedy algorithm has optimal maximum lateness L.

Proof.

- Previous Lemma proves that an optimal schedule with no inversions exists.
- By the Lemma before, all schedule with no inversions have the same maximum lateness
- By construction our algorithm has no inversions, hence it has the same maximum lateness as an optimal schedule

□

The optimal Offline Caching problem

This problem goes also under the name of *Cache maintenance problem*.

The story:

You're working on a long research paper and your draconian librarian will only allow you to have 8 books checked out at once.
You know that you'll probably need more than this over the course of working on the paper, but at any point in time, you'd like to have ready access to the 8 books that are most relevant at that time.

How should you decide which books to check out and when should you return some in exchange for others, to **minimize** the number of times you have to exchange a book at the library?

Memory hierarchies

The problem:

There is a small amount of data that can be accessed very quickly, and a large amount of data that requires more time to access. You must decide which pieces of data to have close at hand.

Applications:

- Computer systems: CPU Cache, RAM, Hard Drive
- Computer networks: Web Caching, Content Caching
- End-host systems: Local Caching at the browser

Caching:

General term for the process of storing a small amount of data in a fast memory so as to reduce the amount of time spent in interacting with a slow memory.

Optimal caching: where is the problem?

For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache.

Definition: cache hit

A cache hit occurs when data is readily accessible from the cache.

A **cache maintenance** algorithm determines what to keep in the cache and what to *evict* from the cache when new data needs to be brought in.

Optimal caching: problem definition (1)

The problem:

We consider a set U of n pieces of data stored in a *main memory*. We also have a faster memory, the **cache**, that can hold $k < n$ pieces of data at any time. We will assume that the cache initially holds some k items.

A sequence of data items $D = d_1, d_2, ...d_m$ drawn from U is presented to us (offline caching), and in processing them we must decide at all times which k items to keep in the cache.

Optimal caching: problem definition (2)

Definition: cache miss

When item d_i is present, we have a **hit**, otherwise we are required to bring it from main memory and we have a **miss.**

The problem: continued

If the cache is full, we need to *evict* some other piece that is currently in the cache to make room for d_i.

Optimal caching: problem definition (3)

Hence, on a particular sequence of memory references, a cache maintenance algorithm determines an **eviction schedule**, specifying which items should be evicted from the cache at which point in the sequence

This determines:

- The contents of the cache
- The number of misses over time

Our objective:

We want to design an algorithm that determines an eviction schedule that **minimizes** the number of misses.

Optimal offline caching: illustrative example

Ex: k = 2, initial cache = ab,

requests: a, b, c, b, c, a, a, b.

Optimal eviction schedule: 2 cache misses.

a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b
requests	cache	

Optimal offline caching: designing the algorithm

In 1960, Les Belady showed that the following simple rule will always incur the minimum number of misses:

```
When $d_i$ needs to be brought into the cache
evict the item that is needed the farthest into the future.
```

We call this the *Farthest-in-Future Algorithm*: when it's time to evict something, we look at the next time that each item in the cache will be referenced, and choose the one for which this is as late as possible.

Optimal offline caching: observations (1)

Though this is a very natural algorithm, the fact that is optimal on all sequences is somewhat more subtle than it first appears.

- Why evict the item that is needed farthest in the future rather than, for example, the one that will be used least frequently in the future?
- Moreover, consider the sequence: $a, b, c, d, a, d, e, a, b, d, c$, with $k = 3$ and items $\{a, b, c\}$ initially in the cache
- The FIF algorithm will produce a schedule S that evicts c on the 4^{th} step and b on the 7^{th} step
- We can imagine a schedule S' that evicts b on the 4^{th} step and c on the 7^{th} with the same number of misses

Optimal offline caching: observations (2)

- So it appears that there are also other schedules that produce the same result as the FIF algorithm!
- However, looking more carefully, it doesn't really matter whether b or c is evicted at the 4^{th} step, since the other one should be evicted at the 7^{th}
- So, given a schedule where b is evicted first, we can **swap** the choices of b and c without changing the cost

These observations lead to what follows: we will use again an *exchange argument*.

Optimal offline caching: some definitions

Definition: reduced schedule

The kind of algorithm we've been considering so far produces a schedule that only bring an item *d* into the cache in a step *i* if there is a request to *d* in step *i*, and *d* is not already in the cache.

We call such a schedule a **reduced** schedule: it does the minimal amount of work necessary in a given step.

In general one could imagine an algorithm that produces a schedule that is not reduced, by bringing in items in steps when they are not requested

Optimal offline caching: analyzing the FIF algorithm (1)

Proposition:

For every non-reduced schedule, there is an equally good reduced schedule.

Proof.

- Let S be a schedule that may be non-reduced
- Define $\hat{S}$, the reduction of S as follows:
  - In any step i, where S brings in an item d that has not been requested, $\hat{S}$ "pretends" to do this but actually d never leaves the main memory.
  - $\hat{S}$ brings d in the cache in the next step j after this in which d is requested
- In this way, the cache miss incurred by $\hat{S}$ in step j can be charged to the earlier cache operation performed by S in step i, when it brought in d.

□

Optimal offline caching: analyzing the FIF algorithm (2)

Proposition:

$\hat{S}$ is a reduced schedule that brings in at most as many items as the schedule S.

Note:

For any reduced schedule, the number of items that are brought in is exactly the number of misses.

PART II: DIVIDE AND CONQUER

Discussion (1)

- Typical example that you all know: `Sorting`
- Class of techniques in which one breaks the input into several parts, solve the problem in each part recursively, and then combines the solutions to these sub-problems into an overall solution.

Discussion (2)

Informally speaking:

- These techniques are useful when the problem is already polynomial;

- In general [Refer to Algorithm Design, Kleinberg- Tardos] these techniques decrease the worst case performance to linear or sub-linear

PART III: DYNAMIC PROGRAMMING

Introduction

- We began this lecture with the study of greedy algorithms, which are in some sense the most natural algorithms, they are myopic and in some cases (difficult to prove) they yield optimal solutions
- We've skipped divide and conquer basic techniques. There are some advanced concepts that allow to reduce exponential execution time down to polynomial time

Dynamic Programming

The basic idea comes from divide an conquer, and is essentially the opposite of the greedy strategy: one implicitly explores the space of all possible solutions, by decomposing things into a series of *sub-problems*, and then building up correct solutions to larger and larger sub-problems.

Examples

These are few examples of problems that can be effectively addressed using dynamic programming:

- Weighted Interval Scheduling
- Shortest Paths and Distance Vector Protocols
- Knapsack problems

Knapsack Problems (1)

Definition (informal)

It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag. Given a set of items, each with a cost and a value, determine the number of each item to include in a collection so that the total cost is less than a given limit and the total value is as large as possible.

The decision problem form of the knapsack problem is the question: “can a value of at least V be achieved without exceeding the cost C?”

Knapsack Problems (2)

Definitions:

We have n kinds of items, $\{1, ..., n\}$. Each item j has a value p_j and a weight w_j. The maximum weight that we can carry in the bag is W.

0-1 Knapsack problem

It is a special case of the original knapsack problem in which each item of input cannot be subdivided to fill a container in which that input partially fits.

The 0-1 knapsack problem restricts the number of each kind of item, x_j, to zero or one.

0-1 Knapsack Problems

Mathematically the 0-1-knapsack problem is :

The problem:

$$\begin{array}{cc} \text{maximize} & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} & \sum_{j=1}^{n} w_j x_j \leq W \\ x_j = 0 \text{ or } 1 & \\ j = 1, ..., n & \end{array}$$

The decision version of the knapsack problem described above is NP-complete.

Fractional Knapsack Problems (1)

Fractional Knapsack problem

There are n items, $i = \{1, 2, ..., n\}$. Item i has weight $w_i > 0$ and value $p_i > 0$. You can carry a maximum weight of W in a knapsack.

Items can be broken into smaller pieces, so you may decide to carry only a fraction x_i of object i, where $0 \leq x_i \leq 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i p_i$ to the value of the load.

Fractional Knapsack Problems (2)

Mathematically, the problem is as follows:

The problem:

$$
\begin{array}{cc}
\text{maximize} & \sum_{j=1}^{n} p_j x_j \\
\text{subject to} & \sum_{j=1}^{n} w_j x_j \leq W \\
0 \leq x_j \leq 1 & \\
j = 1, ..., n &
\end{array}
$$

Fractional Knapsack Problems (3)

Observation:

It is clear that an optimal solution must fill the knapsack exactly, for otherwise we could add a fraction of one of the remaining objects and increase the value of the load.

Thus in an optimal solution:

$$\sum_{j=1}^{n} w_j x_j = W$$

Greedy Algorithm

```
Input: $w, v, W$
for $i = 1$ to $n$ do
    $x[i] = 0$
end
$weight = 0$
while $weight < W$ do
    $i =$ best remaining item
    if $weight + w[i] \leq W$ then
        $x[i] = 1$
        $weight = weight + w[i]$
    else
        $x[i] = (W - weight)/w[i]$
        $weight = W$
    end
end
return $x$
```

Greedy Algorithm: Sketch Analysis

- If the items are already sorted into decreasing order of p_i/w_i, then the while-loop takes a time in $O(n)$
- Therefore, the total time including the sort is in $O(nlogn)$

- If we keep the items in heap with largest p_i/w_i at the root
- Creating the heap takes $O(n)$ time
- While-loop now takes $O(logn)$ time (since heap property must be restored after the removal of root)
- Observation: although this data structure does not alter the worst-case execution time, it may be faster if only a small number of items are needed to fill the knapsack