Data Structures and Algorithms

Lecture 12: Algorithm Design

Algorithm Design

To now, we have examined a number of data structures and algorithms to manipulate them

We have seen examples of efficient strategies

- Divide and conquer
  - Binary search
  - Depth-first tree traversals
  - Merge sort
  - Quicksort
- Greedy algorithms
  - Prim's algorithm
  - Kruskal's algorithm
  - Dijkstra's algorithm

Algorithm Design

We will now examine a number of strategies which may be used in the design of algorithms, including:

- Greedy algorithms
- Divide-and-conquer algorithms
- Dynamic programming
- Backtracking algorithms
- Stochastic algorithms

Algorithm Design

When searching for a solution, we may be interested in two types:

- Either we are looking for the *optimal* solution, or,
- We are interested in a solution which is *good enough*, where good enough is defined by a set of parameters

Algorithm Design

For many of the strategies we will examine, there will be certain circumstances where the strategy can be shown to result in an optimal solution

In other cases, the strategy may not be guaranteed to do so well

Algorithm Design

Any problem may usually be solved in multiple ways

The simplest to implement and most difficult to run is *brute force*

- We consider all possible solutions, and find that solution which is optimal

Algorithm Design

Brute force techniques often take too much time to run

We may use brute-force techniques to show that solutions found through other algorithms are either optimal or close-to-optimal

Algorithm Design

With brute force, we consider all possible solutions

Most other techniques build solutions, thus, we require the following definitions

Definition:

- A *partial solution* is a solution to a problem which could possibly be extended
- A *feasible solution* is a solution which satisfies any given requirements

Algorithm Design

Thus, we would say that a brute-force search tests all feasible solutions

Most techniques will build feasible solutions from *partial solutions* and thereby test only a subset of all possible feasible solutions

Algorithm Design

It may be possible in some cases to have *partial solutions* which are acceptable (that is, feasible) solutions to the problem

In other cases, partial solutions may be unacceptable, and therefore we must continue until we reach a feasible solution

Algorithm Design

We will look at two problems:

- the first requires an exact (optimal) solution,
- the second requires only an approximately optimal solution

In the second case, it would be desirable, but not necessary, to find the optimal solution

Example 1: Sudoku game

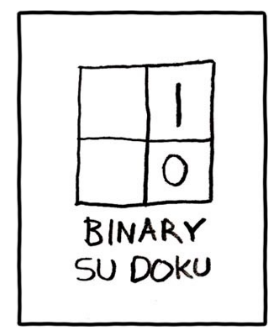
For example, consider the game of Sudoku

The rules are:

- each number must appear once in each row, column, and 3 × 3 outlined square

You are given some initial numbers, and if they are chosen appropriately, there is a unique solution.

8			6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5



http://xkcd.com/74/

Example 1: Sudoku game

Using brute force, we could try every possible solution, and discard those which do not satisfy the conditions

8	1	1	6	1	1	1	1	2
1	4	1	1	5	1	1	1	1
1	1	1	7	1	1	1	1	3
1	9	1	1	1	4	1	1	6
2	1	1	1	1	1	1	1	8
7	1	1	1	1	1	1	1	1
3	1	1	1	1	9	1	1	1
1	1	1	1	8	1	1	1	1
4	1	1	1	1	2	1	1	5

8	1	1	6	1	1	1	1	2
1	4	1	1	5	1	1	1	1
1	1	1	7	1	1	1	1	3
1	9	1	1	1	4	1	1	6
2	1	1	1	1	1	1	1	8
7	1	1	1	1	1	1	1	1
3	1	1	1	1	9	1	1	1
1	1	1	1	8	1	1	1	1
4	1	1	1	1	2	1	2	5

This technique would require us to check $9^{61} \approx 1.6 \times 10^{58}$ possible solutions

Example 2: Project management

Suppose you are a manager, and you have 26 weeks or half a year for the next product cycle

You have n possible projects, however, the time required to complete these projects is much greater than 26 weeks

Associated with each possible project are numerous factors:

- The expected completion time
- The expected increase in revenue
- A probability of failure
- Possible future projects which may benefit

Example 2: Project management

Stake holders include:

- Team members
- Marketing
- Other management
- The executive team

You must now decide which projects must be chosen so as to satisfy the schedule

- It must be justifiable

Example 2: Project management

In this case, it is almost impossible to come up with a *optimal* choice of projects, however, you are required to come up with an appropriate solution

We will see how an appropriate choice of algorithm may lead us towards a reasonably optimal solution

Example 2: Project management

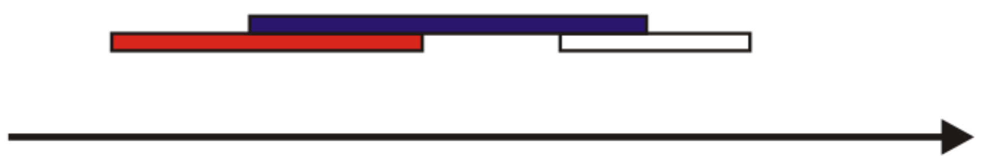
In this case, any sub-set of the n projects forms a *partial* solution

A partial solution is a feasible solution if the sum of the expected completion times is less than six months

Example 3: Interval scheduling

Another case we will look at is interval scheduling:

- Given n processes, all of which must be run at specific times, maximize the number of processes that are run

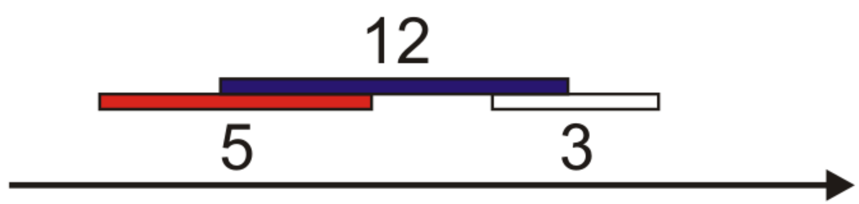


This has a reasonably simple solution that we will see later

Example 3: Interval scheduling

However, if you modify the problem:

- Given n processes, all of which must be run at specific times and where each is given a specific *weight*, maximize the total weight of the processes that are run



Greedy algorithms

- This topic will cover greedy algorithms:
  - Definitions
  - Examples
    - Making change
    - Prim's and Dijkstra's algorithm
  - Other examples

Greedy algorithms

Suppose it is possible to build a solution through a sequence of partial solutions

- At each step, we focus on one particular partial solution and we attempt to extend that solution
- Ultimately, the partial solutions should lead to a feasible solution which is also optimal

Making change

Consider this commonplace example:

- Making the exact change with the ***minimum*** number of coins
- Consider the Euro denominations of 1, 2, 5, 10, 20, 50 cents
- Stating with an empty set of coins, add the largest coin possible into the set which does not go over the required amount

Making change

To make change for €0.72:

- Start with €0.50

Total €0.50

Making change

To make change for €0.72:

- Start with €0.50
- Add a €0.20

Total €0.70

Making change

To make change for €0.74:

- Start with €0.50
- Add a €0.20
- Skip the €0.10 and the €0. 05 but add a €0.02

Total €0.72

Making change

Notice that each digit can be worked with separately

- The maximum number of coins for any digit is three
- Thus, to make change for anything less than €1 requires at most six coins
- The solution is optimal

Making change

Does this strategy always work?

- What if our coin denominations grow quadraticly?

Consider 1, 4, 9, 16, 25, 36, and 49 dumbledores

Reference: J.K. Rowlings, *Harry Potter*, Raincoast Books, 1997.

Making change

Using our algorithm, to make change for 72 dumbledores, we require six coins:

$$72 = 49 + 16 + 4 + 1 + 1 + 1$$

Making change

The optimal solution, however, is **two** 36 dumbledore coins

Definition

A greedy algorithm is an algorithm which has:

- A set of partial solutions from which a solution is built
- An *objective function* which assigns a value to any partial solution

Then given a partial solution, we

- Consider possible extensions of the partial solution
- Discard any extensions which are not feasible
- Choose that extension which minimizes the object function

This continues until some criteria has been reached.

Optimal example

Prim's algorithm is a greedy algorithm:

- Any connected sub-graph of k vertices and $k - 1$ edges is a partial solution
- The value to any partial solution is the sum of the weights of the edges

Then given a partial solution, we

- Add that edge which does not create a cycle in the partial solution and which minimizes the increase in the total weight
- We continue building the partial solution until the partial solution has n vertices
- An optimal solution is found.

Optimal example

Dijkstra's algorithm is a greedy algorithm:

- A subset of k vertices and known the minimum distance to all k vertices is a partial solution

Then given a partial solution, we

- Add that edge which is smallest which connects a vertex to which the minimum distance is known and a vertex to which the minimum distance is not known
- We define the distance to that new vertex to be the distance to the known vertex plus the weight of the connecting edge
- We continue building the partial solution until either:
  - The minimum distance to a specific vertex is known, or
  - The minimum distance to all vertices is known
- An optimal solution is found

Optimal and sub-optimal examples

Our coin change example is greedy:

- Any subset of k coins is a partial solution
- The value to any partial solution is the sum of the values

Then given a partial solution, we

- Add that coin which maximizes the increase in value without going over the target value
- We continue building the set of coins until we have reached the target value

An optimal solution is found with euros, but not with the *quadratic* dumbledore coins.

Unfeasible example

In some cases, it may be possible that not even a feasible solution is found

- Consider the following greedy algorithm for solving Sudoku:
- For each empty square, starting at the top-left corner and going across:
  - Fill that square with the smallest number which does not violate any of our conditions
  - All feasible solutions have equal weight

Unfeasible example

Let's try this example the previously seen Sudoku square:

8			6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible example

Neither 1 nor 2 fits into the first empty square, so we fill it with 3

8	3		6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible example

The second empty square may be filled with 1

8	3	1	6					2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible example

And the 3rd empty square may be filled with 4

8	3	1	6	4				2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible example

At this point, we try to fill in the 4^{th} empty square

8	3	1	6	4	?			2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Unfeasible example

Unfortunately, all nine numbers 1 – 9 already appear in such a way to block it from appearing in that square

- There is no known greedy algorithm which finds the one feasible solution

8	3	1	6	4	?			2
	4			5			1	
			7					3
	9				4			6
2								8
7				1			5	
3					9			
	1			8			9	
4					2			5

Project management
0/1 knapsack problem

Situation:

- The next cycle for a given product is 26 weeks
- We have **ten** possible projects which could be completed in that time, each with an expected number of weeks to complete the project and an expected increase in revenue

Project management
0/1 knapsack problem

Objective:

- As project manager, choose those projects which can be completed in the required amount of time which maximizes revenue

This is also called the 0/1 knapsack problem

- You can place n items in a knapsack where each item has a value in rupees and a weight in kilograms
- The knapsack can hold a maximum of m kilograms

Project management
0/1 knapsack problem

The projects:

Product ID	Completion Time (wks)	Expected Revenue (1000 $)
A	15	210
B	12	220
C	10	180
D	9	120
E	8	160
F	7	170
G	5	90
H	4	40
J	3	60
K	1	10

Project management
0/1 knapsack problem

Let us first try to find an optimal schedule by trying to be as productive as possible during the 26 weeks:

- we will start with the projects in order from most time to least time, and at each step, select the longest-running project which does not put us over 26 weeks
- we will be able to fill in the gaps with the smaller projects

Project management
0/1 knapsack problem

Greedy-by-time (make use of all 26 wks):

- Project A:15 wks
- Project C:10 wks
- Project J: 1 wk

Total time: 26 wks

Expected revenue:

$400 000

Product ID	Completion Time (wks)	Expected Revenue (1000 $)
A	**15**	**210**
B	**12**	220
C	**10**	**180**
D	**9**	120
E	**8**	160
F	**7**	170
G	**5**	90
H	**4**	40
I	**3**	60
J	**1**	**10**

Project management 0/1 knapsack problem

Next, let us attempt to find an optimal schedule by starting with the most :

- we will start with the projects in order from most time to least time, and at each step, select the longest-running project which does not put us over 26 weeks
- we will be able to fill in the gaps with the smaller projects

Project management
0/1 knapsack problem

Greedy-by-revenue (best-paying projects):

- Project B:$220K
- Project C:$180K
- Project H:$ 60K
- Project K:$ 10K

Product ID	Completion Time (wks)	Expected Revenue (1000 $)
B	12	**220**
A	15	**210**
C	10	**180**
F	7	**170**
E	8	**160**
D	9	**120**
G	5	**90**
J	3	**60**
H	4	**40**
K	1	**10**

Total time: 26 wks

Expected revenue:

$470 000

Project management
0/1 knapsack problem

Unfortunately, either of these techniques focuses on projects which have high projected revenues or high run times

What we really want is to be able to complete those jobs which pay the most per unit of development time

Thus, rather than using development time or revenue, let us calculate the expected revenue per week of development time

Project management
0/1 knapsack problem

This is summarized here:

Product ID	Completion Time (wks)	Expected Revenue (1000 $)	Revenue Density ($ / wk)
A	15	210	**14 000**
B	12	220	**18 333**
C	10	180	**18 000**
D	9	120	**13 333**
E	8	160	**20 000**
F	7	170	**24 286**
G	5	90	**18 000**
H	4	40	**10 000**
J	3	60	**20 000**
K	1	10	**10 000**

Project management
0/1 knapsack problem

Greedy-by-revenue-density:

- Project F:$24 286/wk
- Project E:$20 000/wk
- Project J:$20 000/wk
- Project G:$18 000/wk
- Project K:$10 000/wk

Total time: 24 wks

Expected revenue:

$490 000

Bonus: 2 weeks for bug fixing

Product ID	Completion Time (wks)	Expected Revenue (1000 $)	Revenue Density ($/wk)
F	7	170	**24 286**
E	8	160	**20 000**
J	3	60	**20 000**
B	12	220	**18 333**
C	10	180	**18 000**
G	5	90	**18 000**
A	15	210	**14 000**
D	9	120	**13 333**
H	4	40	**10 000**
K	1	10	**10 000**

Project management
0/1 knapsack problem

Using brute force, we find that the optimal solution is:

- Project C:$180 000
- Project E:$170 000
- Project F:$150 000
- Project K:$ 10 000

Total time: 26 wks

Expected revenue:

$520 000

Product ID	Completion Time (wks)	Expected Revenue (1000 $)	Revenue Density ($/wk)
A	15	210	14 000
B	12	220	18 333
C	10	180	18 000
D	9	120	13 333
E	8	160	20 000
F	7	170	24 286
G	5	90	18 000
H	4	40	10 000
J	3	60	20 000
K	1	10	10 000

Project management
0/1 knapsack problem

In this case, the greedy-by-revenue-density came closest to the optimal solution:

Algorithm	Expected Revenue
Greedy-by-time	\$400 000
Greedy-by-expected revenue	\$470 000
Greedy-by-revenue density	\$490 000
Brute force	\$520 000

- The run time is $\Theta(n \ln(n))$ — the time required to sort the list
- Later, we will see a dynamic program for finding an optimal solution with one additional constraint

Project management
0/1 knapsack problem

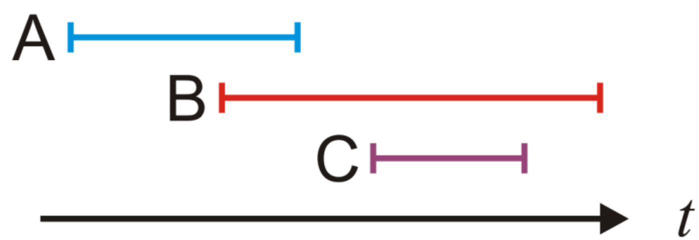
Of course, in reality, there are numerous other factors affecting projects, including:

- Flexible deadlines (if a delay by a week would result in a significant increase in expected revenue, this would be acceptable)
- Probability of success for particular projects
- The requirement for *banner* projects
  - Note that greedy-by-revenue-density had none of the larger projects

Interval scheduling

Suppose we have a list of processes, each of which must run in a given time interval: e.g.,

- process A must run during 2:00-5:00
  process B must run during 4:00-9:00
  process C must run during 6:00-8:00



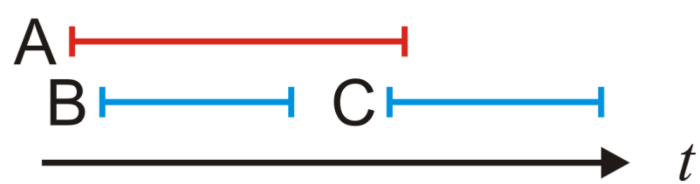
Interval scheduling

Suppose we want to maximize the *number* of processes that are run

In order to create a greedy algorithm, we must have a fast selection process which quickly determines which process should be run next

The first thought may be to always run that process that is next ready to run

- A little thought, however, quickly demonstrates that this fails



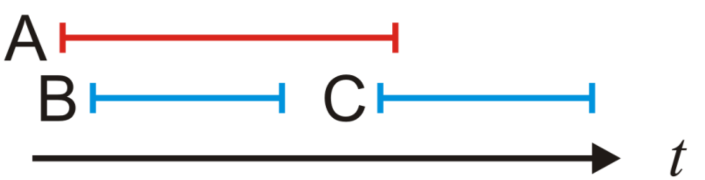
- The worst case would be to only run 1 out of n possible processes when $n - 1$ processes could have been run

Interval scheduling

To maximize the *number* of processes that are run, we should
trying to free up the processor as quickly as possible

- Instead of looking at the start times, look at the end times
- At any time that the processor is available, select that process with the earliest end time: the *earliest-deadline-first* algorithm

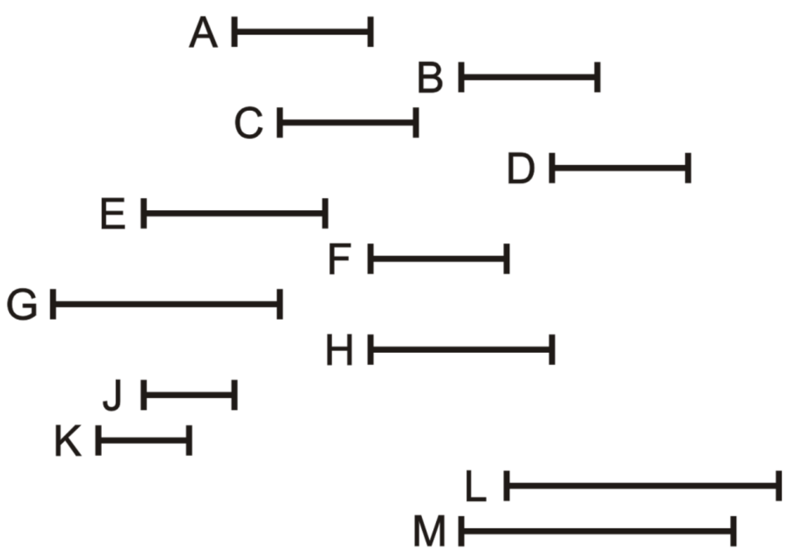
In this example, Process B is the first to start, and then Process C follows:



Interval scheduling

Consider the following list of 12 processes together with the time interval during which they must be run

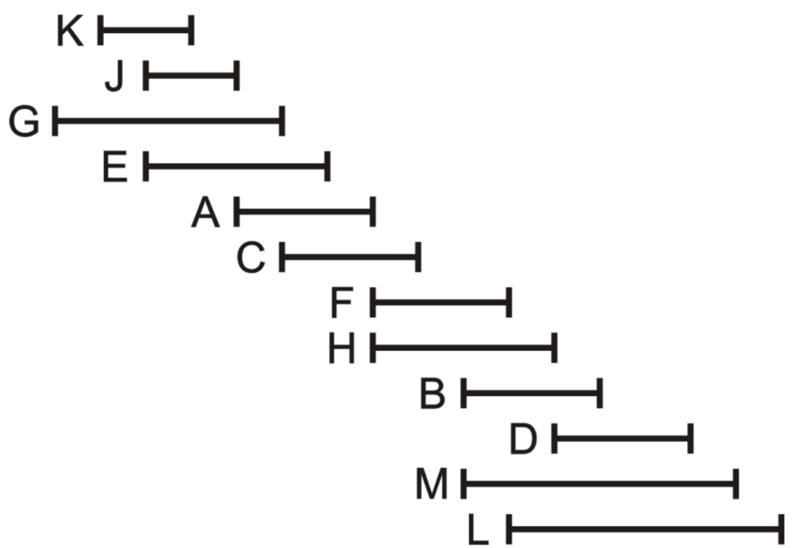
- Find the optimal schedule with the earliest-deadline-first greedy algorithm



Process	Interval
A	5 - 8
B	10 - 13
C	6 - 9
D	12 - 15
E	3 - 7
F	8 - 11
G	1 - 6
H	8 - 12
J	3 - 5
K	2 - 4
L	11 - 16
M	10 - 15

Interval scheduling

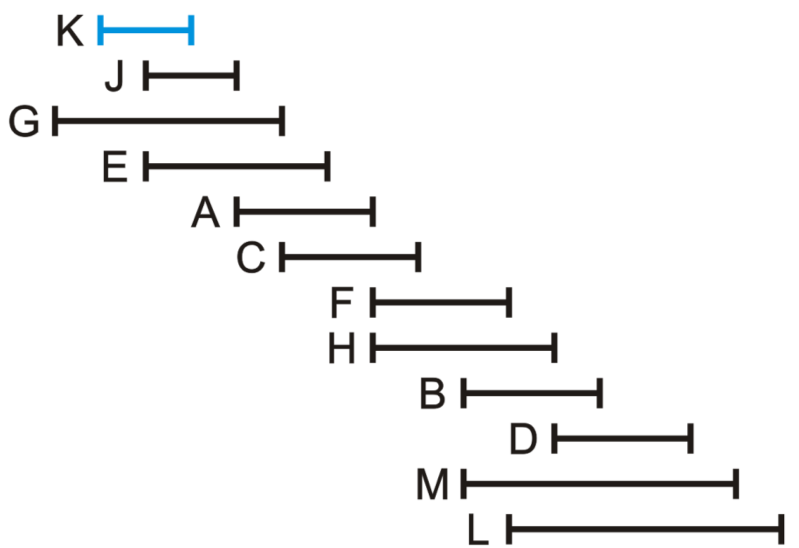
In order to simplify this, sort the processes on their end times



Process	Interval
K	2 - 4
J	3 - 5
G	1 - 6
E	3 - 7
A	5 - 8
C	6 - 9
F	8 - 11
H	8 - 12
B	10 - 13
D	12 - 15
M	10 - 15
L	11 - 16

Interval scheduling

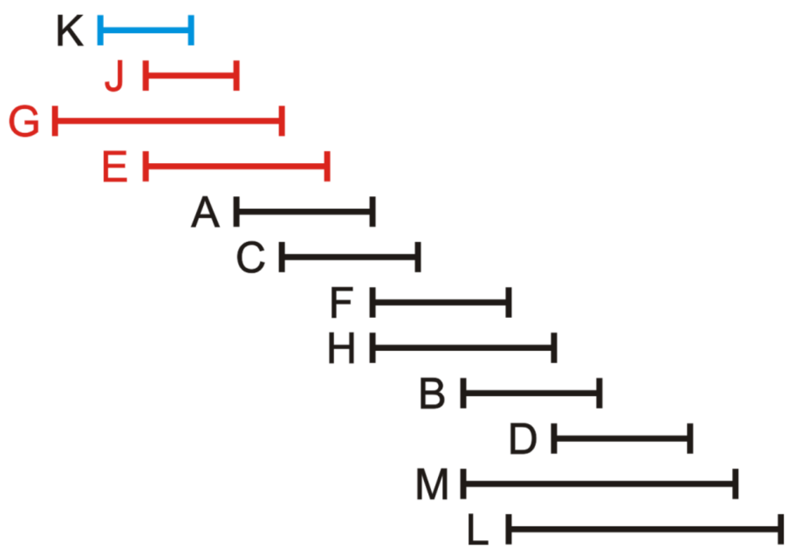
To begin, choose Process K



Process	Interval
K	2 - 4
J	3 - 5
G	1 - 6
E	3 - 7
A	5 - 8
C	6 - 9
F	8 - 11
H	8 - 12
B	10 - 13
D	12 - 15
M	10 - 15
L	11 - 16

Interval scheduling

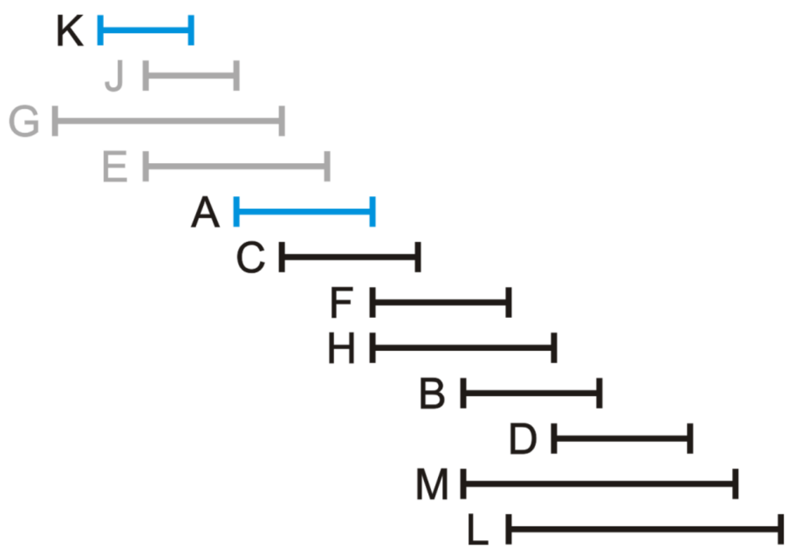
At this point, Process J, G and E can no longer be run



Process	Interval
K	2 - 4
J	3 - 5
G	1 - 6
E	3 - 7
A	5 - 8
C	6 - 9
F	8 - 11
H	8 - 12
B	10 - 13
D	12 - 15
M	10 - 15
L	11 - 16

Interval scheduling

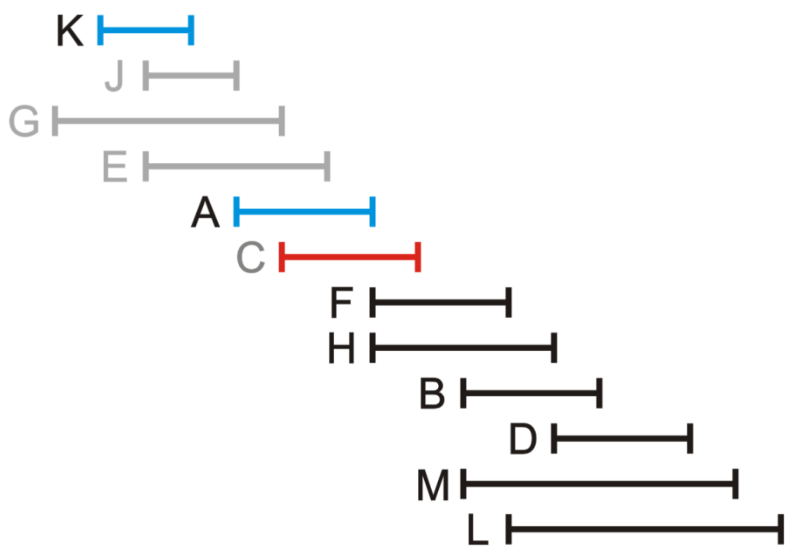
Next, run Process A



Process	Interval
K	**2 - 4**
J	3 - 5
G	1 - 6
E	3 - 7
A	**5 - 8**
C	6 - 9
F	8 - 11
H	8 - 12
B	10 - 13
D	12 - 15
M	10 - 15
L	11 - 16

Interval scheduling

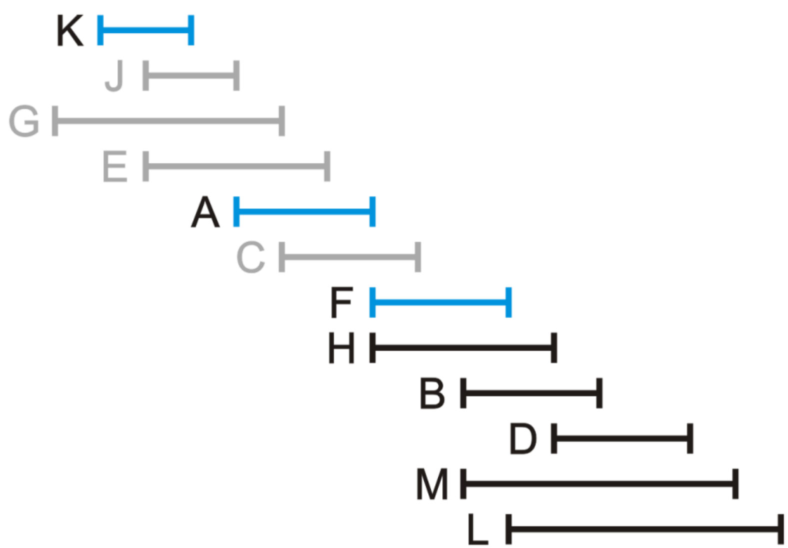
We can no longer run Process C



Process	Interval
K	2 - 4
J	3 - 5
G	1 - 6
E	3 - 7
A	5 - 8
C	6 - 9
F	8 - 11
H	8 - 12
B	10 - 13
D	12 - 15
M	10 - 15
L	11 - 16

Interval scheduling

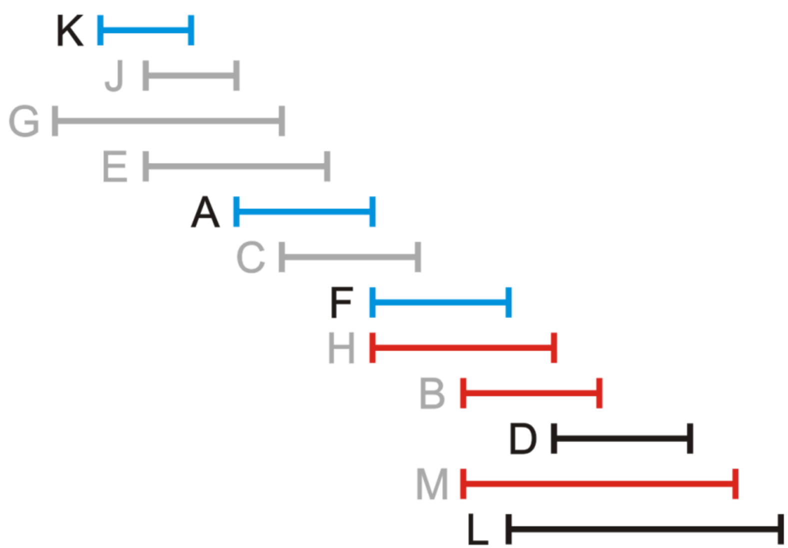
Next, we can run Process F



Process	Interval
K	**2 – 4**
J	3 – 5
G	1 – 6
E	3 – 7
A	**5 – 8**
C	6 – 9
F	**8 – 11**
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling

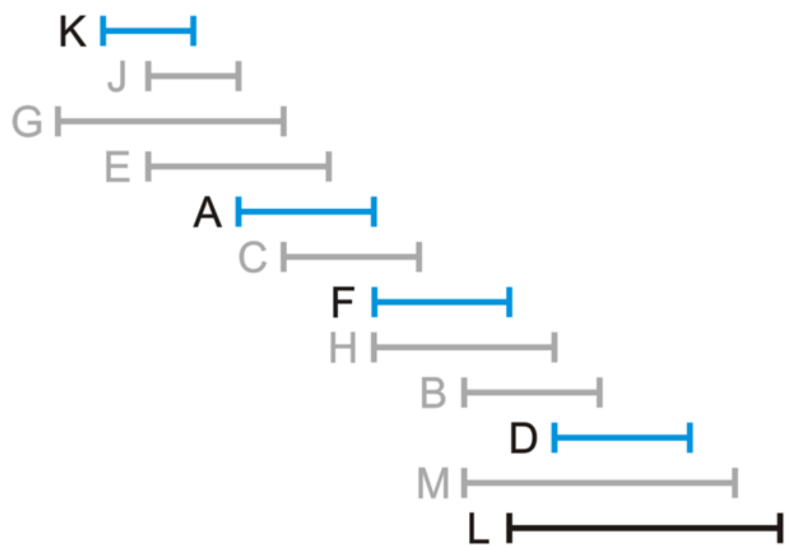
This restricts us from running Processes H, B and M



Process	Interval
K	2 - 4
J	3 - 5
G	1 - 6
E	3 - 7
A	5 - 8
C	6 - 9
F	8 - 11
H	8 - 12
B	10 - 13
D	12 - 15
M	10 - 15
L	11 - 16

Interval scheduling

The next available process is D

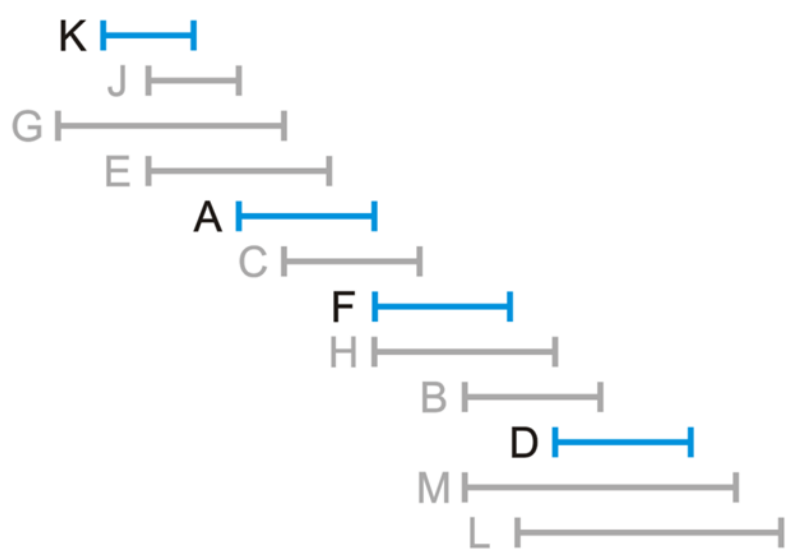


Process	Interval
K	**2 – 4**
J	3 – 5
G	1 – 6
E	3 – 7
A	**5 – 8**
C	6 – 9
F	**8 – 11**
H	8 – 12
B	10 – 13
D	**12 – 15**
M	10 – 15
L	11 – 16

Interval scheduling

The prevents us from running Process L

- We are therefore finished



Process	Interval
K	**2 – 4**
J	3 – 5
G	1 – 6
E	3 – 7
A	**5 – 8**
C	6 – 9
F	**8 – 11**
H	8 – 12
B	10 – 13
D	**12 – 15**
M	10 – 15
L	11 – 16

Application: Interval scheduling

We have scheduled four processes

- The selection may not be unique

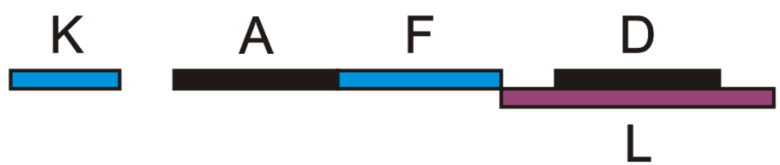
Once the processes are sorted, the run time is linear—we simply look ahead to find the next process that can be run

- Thus, the run time is the run time of sorting

Process	Interval
K	**2 – 4**
J	3 – 5
G	1 – 6
E	3 – 7
A	**5 – 8**
C	6 – 9
F	**8 – 11**
H	8 – 12
B	10 – 13
D	**12 – 15**
M	10 – 15
L	11 – 16

Application: Interval scheduling

For example, we could have chosen Process L

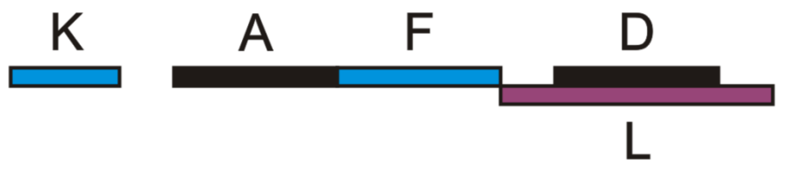


In this case, processor usage would go up, but no significance is given to that criteria

Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Application: Interval scheduling

We could add weights to the individual processes



- The weights could be the duration of the processes—maximize processor usage
- The weights could be revenue gained from the performance—maximize revenue

Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Summary of greedy algorithms

We have seen the algorithm-design technique, namely greedy algorithms

- For some problems, appropriately-designed greedy algorithms may find either optimal or near-optimal solutions
- For other problems, greedy algorithms may a poor result or even no result at all

Their desirable characteristic is speed

Divide-and-conquer algorithms

We have seen four divide-and-conquer algorithms:

- Binary search
- Depth-first tree traversals
- Merge sort
- Quick sort

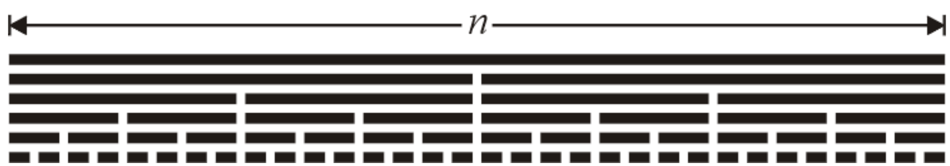
The steps are:

- A larger problem is broken up into smaller problems
- The smaller problems are recursively
- The results are combined together again into a solution

Divide-and-conquer algorithms

For example, merge sort:

- Divide a list of size n into $b = 2$ sub-lists of size $n/2$ entries
- Each sub-list is sorted recursively
- The two sorted lists are merged into a single sorted list



Divide-and-conquer algorithms

More formally, we will consider only those algorithms which:

- Divide a problem into b sub-problems, each approximately of size n/b
  - Up to now, $b = 2$
- Solve $a \geq 1$ of those sub-problems recursively
  - Merge sort and tree traversals solved $a = 2$ of them
  - Binary search solves $a = 1$ of them
- Combine the solutions to the sub-problems to get a solution to the overall problem

Divide-and-conquer algorithms

With the three problems we have already looked at we have looked at two possible cases for $b = 2$:

- Merge sort $b = 2$ $a = 2$
- Depth-first traversal $b = 2$ $a = 2$
- Binary search $b = 2$ $a = 1$

Problem: the first two have different run times:

- Merge sort $\Theta(n \ln(n))$
- Depth-first traversal $\Theta(n)$

Divide-and-conquer algorithms

Thus, just using a divide-and-conquer algorithm does not solely determine the run time

We must also consider

- The effort required to divide the problem into two sub-problems
- The effort required to combine the two solutions to the sub-problems

Divide-and-conquer algorithms

For merge sort:

- Division is quick (find the middle): $\Theta(1)$
- Merging the two sorted lists into a single list is a $\Theta(n)$ problem

For a depth-first tree traversal:

- Division is also quick: $\Theta(1)$
- A return-from-function is preformed at the end which is $\Theta(1)$

For quick sort (assuming division into two):

- Dividing is slow: $\Theta(n)$
- Once both sub-problems are sorted, we are finished: $\Theta(1)$

Divide-and-conquer algorithms

Thus, we are able to write the expression as follows:

- Binary search:

  $\mathbf{\Theta}(\ln(n))$

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ \mathrm{T}\left(\frac{n}{2}\right) + \mathbf{\Theta}(1) & n > 1 \end{cases}$$

- Depth-first traversal:

  $\mathbf{\Theta}(n)$

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ 2\mathrm{T}\left(\frac{n}{2}\right) + \mathbf{\Theta}(1) & n > 1 \end{cases}$$

- Merge/quick sort:

  $\mathbf{\Theta}(n \ln(n))$

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ 2\mathrm{T}\left(\frac{n}{2}\right) + \mathbf{\Theta}(n) & n > 1 \end{cases}$$

In general, we will assume the work done combined work is of the form $\mathbf{O}(n^k)$

Divide-and-conquer algorithms

Thus, for a general divide-and-conquer algorithm which:

- Divides the problem into b sub-problems
- Recursively solves a of those sub-problems
- Requires $\mathbf{O}(n^k)$ work at each step requires

has a run time

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ a\,\mathrm{T}\left(\frac{n}{b}\right) + \mathbf{O}\left(n^k\right) & n > 1 \end{cases}$$

Note: we assume a problem of size $n = 1$ is solved...

Summary of divide-and-conquer algo.

Divide-and-conquer algorithms:

- If the amount of work being done at each step to either sub-divide the problem or to recombine the solutions dominates, then this is the run time of the algorithm: $\mathbf{O}(n^k)$
- If the problem is being divided into many small sub-problems ($a > b^k$) then the number of sub-problems dominates: $\mathbf{O}(n^{\log_b(a)})$
- In between, a little more (logarithmically more) work must be done

Dynamic programming

- This topic will cover dynamic programming:
  - Definitions
  - An Example
    - Fibonacci numbers
  - Other applications
    - Interval scheduling
    - Project management - 0/1 knapsack problem

Dynamic programming

To begin, the word *programming* is used by mathematicians to describe a set of rules which must be followed to solve a problem

- Thus, *linear programming* describes sets of rules which must be solved a linear problem
- In our context, the adjective *dynamic* describes how the set of rules works

Dynamic programming

Dynamic programming is distinct from divide-and-conquer, as the divide-and-conquer approach works well if the sub-problems are essentially unique

- Storing intermediate results would only waste memory

If sub-problems re-occur, the problem is said to have *overlapping sub-problems*

Fibonacci numbers

Consider this function:

```
double F( int n ) {
    return ( n <= 1 ) ? 1.0 : F(n - 1) + F(n - 2);
}
```

The run-time of this algorithm is

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & n \le 1 \\ \mathrm{T}(n-1) + \mathrm{T}(n-2) + \Theta(1) & n > 1 \end{cases}$$

Fibonacci numbers

Consider this function calculating Fibonacci numbers:

```
double F( int n ) {
    return ( n <= 1 ) ? 1.0 : F(n - 1) + F(n - 2);
}
```

The runtime is similar to the definition of Fibonacci numbers:

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & n \le 1 \\ \mathrm{T}(n-1) + \mathrm{T}(n-2) + \Theta(1) & n > 1 \end{cases} \qquad F(n) = \begin{cases} 1 & n \le 1 \\ F(n-1) + F(n-2) + 1 & n > 1 \end{cases}$$

Therefore, $\mathrm{T}(n) = \Omega(F(n)) = \Omega(\phi^n)$

- In actual fact, $\mathrm{T}(n) = \Theta(\phi^n)$, only $\lim_{n \to \infty} \frac{T(n)}{F(n)} = 2$

Fibonacci numbers

To demonstrate, consider:

```
double F( int n ) {
    return ( n <= 1 ) ? 1.0 : F(n - 1) + F(n - 2);
}
```

```
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    cout.precision( 16 );    // print 16 decimal digits of precision for doubles
                                // 53/lg(10) = 15.95458977...

    for ( int i = 33; i < 100; ++i ) {
        cout << "F(" << i << ") = "
             << F(i) << '\t' << time(0) << endl;
    }

    return 0;
}
```

Fibonacci numbers

The output:

```
F(33) = 5702887120647435
F(34) = 9227465120647435.
F(35) = 14930352120647435
F(36) = 24157817120647435
F(37) = 39088169120647435
F(38) = 63245986120647436
F(39) = 102334155120647436
F(40) = 165580141120647436
F(41) = 267914296120647437
F(42) = 433494437120647438
F(43) = 701408733120647441
F(44) = 1134903170120647446
```

F(33), F(34), and F(35) in 1 s

~1 min to calculate F(44)

Fibonacci numbers

Problem:

- To calculate $F(44)$, it is necessary to calculate $F(43)$ and $F(42)$
- However, to calculate $F(43)$, it is also necessary to calculate $F(42)$
- It gets worse, for example
  - $F(40)$ is called 5 times
  - $F(30)$ is called 620 times
  - $F(20)$ is called 75 025 times
  - $F(10)$ is called 9 227 465 times
  - $F(0)$ is called 433 494 437 times

Surely we don't have to recalculate $F(10)$ almost ten million times…

Fibonacci numbers

Here is a possible solution:

- To avoid calculating values multiple times, store intermediate calculations in a table
- When storing intermediate results, this process is called *memoization*
  - The root is *memo*
- We save (*memoize*) computed answers for possible later reuse, rather than re-computing the answer multiple times

Fibonacci numbers

Once we have calculated a value, can we not store that value and return it if a function is called a second time?

- One solution: use an array
- Another: use an associative hash table

Fibonacci numbers

```
static const int ARRAY_SIZE = 1000;
double * array = new double[ARRAY_SIZE];

array[0] = 1.0;
array[1] = 1.0;

// use 0.0 to indicate we have not yet cal
for ( int i = 2; i < ARRAY_SIZE;
    array[i] = 0.0;
}

int n ) {
    if ( array[n] == 0.0 ) {
        array[n] = F( n – 1 ) + F( n – 2 );
    }

    return array[n];
}
```

Problems: What if you go beyond the end of the array?
What if our problem is not indexed by integers?

Fibonacci numbers

Recall the characteristics of an associative container:

```
template <typename S, typename T>
class Hash_table {
    public:
        // is something stored with the given key?
        bool member( S key ) const;

        // returns value associated with the inserted key
        T retrieve( S key ) const;

        void insert( S key, T value );
        // ...
};
```

Fibonacci numbers

This program uses the Standard Template Library:

```
#include <map>

/* calculate the nth Fibonacci number */
double F( int n ) {
        static std::map<int, double> memo;
                // shared by all calls to F
                // the key is int, the value is double

        if ( n <= 1 ) {
                return 1.0;
        } else {
                if ( memo[n] == 0.0 ) {
                        memo[n] = F(n - 1) + F(n - 2);
                }

                return memo[n];
        }
}
```

Fibonacci numbers

This prints the output up to $F(1476)$:

```
int main() {
        std::cout.precision( 16 );

        for ( int i = 0; i < 1476; ++i ) {
                std::cout << "F(" << i << ") = " << F(i) << std::endl;
        }

        return 0;
}
```

The last two lines are

F(1475) = 1.306989223763399e+308

F(1476) = inf

Exact value: F(1475) = 1306989223763399318036311553802719830983924439074126407260066594601927930704792317402886810877770177210954631549790122762343222469369396471853667063684893626608441474499413484628009227558189696347433489829164249540627441359698656154072764924106537217745906695448014908376491617320959726580646300337933471716 32

Summary of dynamic programming

We have considered the algorithm design strategy of dynamic programming

- Useful when recursive algorithms have overlapping sub-problems
- Storing calculated values allows significant reductions in time
  - Memoization
- More applications
  - Interval scheduling
  - Project management - 0/1 knapsack problem
  - …

Wrape up

- We examined a numer of algorithm design techniques which may, in some circumstances provide either optimal or near-optimal solutions
  - Greedy algorithms
  - Divide-and-conquer algorithms
  - Dynamic programming