Greedy Algorithms

Abhiram Ranade

February 1, 2016

Interval scheduling

Interval scheduling

Input: $S[1..n], F[1..n]$

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Output: Largest subset of jobs no two of which overlap.

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Output: Largest subset of jobs no two of which overlap.

Only one job can execute at a time, execute as many as possible.

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Output: Largest subset of jobs no two of which overlap.

Only one job can execute at a time, execute as many as possible.

Guaranteed correct algorithm: Try all possible subsets, and pick largest.

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Output: Largest subset of jobs no two of which overlap.

Only one job can execute at a time, execute as many as possible.

Guaranteed correct algorithm: Try all possible subsets, and pick largest.

Can we try discovering a greedy strategy using the exchange argument?

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Output: Largest subset of jobs no two of which overlap.

Only one job can execute at a time, execute as many as possible.

Guaranteed correct algorithm: Try all possible subsets, and pick largest.

Can we try discovering a greedy strategy using the exchange argument?

Seems difficult.

Interval scheduling

Input: $S[1..n], F[1..n]$

$S[i], F[i]$: start and finish time of job i

Output: Largest subset of jobs no two of which overlap.

Only one job can execute at a time, execute as many as possible.

Guaranteed correct algorithm: Try all possible subsets, and pick largest.

Can we try discovering a greedy strategy using the exchange argument?

Seems difficult.

Another attempt: can we (greedily) identify one job which we feel must be in the optimum solution?

Different ways of being greedy!

Different ways of being greedy!

Benefit of picking a job:

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

Show this does not work.

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.

- Pick the job which conflicts with fewest other jobs.

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.

- Pick the job which conflicts with fewest other jobs.

  This also does not work!

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.

- Pick the job which conflicts with fewest other jobs.

  This also does not work!

- Pick the job with the earliest starting time.

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.

- Pick the job which conflicts with fewest other jobs.

  This also does not work!

- Pick the job with the earliest starting time.

  Running out of imagination..

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.

- Pick the job which conflicts with fewest other jobs.

  This also does not work!

- Pick the job with the earliest starting time.

  Running out of imagination..

- Pick the job with the earliest finishing time..

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.
- Pick the job which conflicts with fewest other jobs.

  This also does not work!
- Pick the job with the earliest starting time.

  Running out of imagination..
- Pick the job with the earliest finishing time..

  More so...

Different ways of being greedy!

Benefit of picking a job: intrinsically good!

(Potential) cost of picking a job: conflicting jobs cannot be picked.

Two jobs conflict if the intervals overlap.

Some (greedy) ideas

- Pick the job with least duration.

  Show this does not work.

- Pick the job which conflicts with fewest other jobs.

  This also does not work!

- Pick the job with the earliest starting time.

  Running out of imagination..

- Pick the job with the earliest finishing time..

  More so...

Earliest finishing time works!

Next.

Earliest finishing job must be in some optimal solution

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.

Suppose earliest finishing job (job j) is not in it.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.
So they must start before f_j.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.
So they must start before f_j.
Thus $k, l, \ldots$ overlap with each other at f_j

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.
So they must start before f_j.
Thus $k, l, \ldots$ overlap with each other at f_j

Jobs in T cannot overlap with each other, so there cannot be many jobs $k, l, \ldots$

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.
So they must start before f_j.
Thus $k, l, \ldots$ overlap with each other at f_j

Jobs in T cannot overlap with each other, so there cannot be many jobs $k, l, \ldots$
At most one job k conflicts with j.

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.
So they must start before f_j.
Thus $k, l, \ldots$ overlap with each other at f_j

Jobs in T cannot overlap with each other, so there cannot be many jobs $k, l, \ldots$
At most one job k conflicts with j.

$T' = T - \{k\} \cup \{j\}$ is a valid solution, $|T'| = |T|$

Earliest finishing job must be in some optimal solution

Proof: Consider an optimal solution T.
Suppose earliest finishing job (job j) is not in it.

Let us try to add j to T.
If no job in T conflicts with j, we will get better solution than optimal.
Thus some jobs in T must conflict with job j. Say jobs $k, l, \ldots$.

Job j has earliest finishing time. So jobs $k, l, \ldots$ finish after f_j.
So they must start before f_j.
Thus $k, l, \ldots$ overlap with each other at f_j

Jobs in T cannot overlap with each other, so there cannot be many jobs $k, l, \ldots$
At most one job k conflicts with j.

$T' = T - \{k\} \cup \{j\}$ is a valid solution, $|T'| = |T|$
Thus we have found an optimal solution which has j.

The rest of the optimal solution

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
j = job with least $F[j]$ must be in some optimal solution

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j.

S', F' define another instance.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
j = job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let S', F' = starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'. O' is a solution for S', F'

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'. O' is a solution for S', F'
$|O'| \leq |T'|$ because T' is optimal for S', F'.

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'. O' is a solution for S', F'
$|O'| \leq |T'|$ because T' is optimal for S', F'.
$|O| = 1 + |O'|$

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'. O' is a solution for S', F'
$|O'| \leq |T'|$ because T' is optimal for S', F'.
$|O| = 1 + |O'| \leq 1 + |T'|$

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
j = job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let S', F' = starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'. O' is a solution for S', F'
$|O'| \leq |T'|$ because T' is optimal for S', F'.
$|O| = 1 + |O'| \leq 1 + |T'| \leq |T|$

The rest of the optimal solution

$S[1..n], F[1..n]$: original instance
$j =$ job with least $F[j]$ must be in some optimal solution
If job j is in solution, jobs that conflict with it cannot be.
Let $S', F' =$ starting,finishing times of jobs except for j and those that conflict with j. S', F' define another instance.

Claim: Let T' denote the optimal solution for S', F'. Then the $T = \{j\} \cup T'$ is an optimal solution for S, F.
Proof:
Suppose an optimal solution O for S, F has more jobs than T.
Since j is guaranteed to be in some optimal solution for S, F, wlog j is present in O. O does not have jobs that conflict with j.
Let $O' = O - \{j\}$. O' also does not have jobs conflicting with j.
O' only contains jobs in S', F'. O' is a solution for S', F'
$|O'| \leq |T'|$ because T' is optimal for S', F'.
$|O| = 1 + |O'| \leq 1 + |T'| \leq |T|$ Contradiction!

Algorithm

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If $n = 1$ return 1

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If $n = 1$ return 1
2. $j =$ integer s.t. F[j] smallest in F[1..n]

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If $n = 1$ return 1
2. $j =$ integer s.t. F[j] smallest in F[1..n]
3. Remove j and all jobs that conflict with j from S,F.

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If $n = 1$ return 1
2. $j =$ integer s.t. F[j] smallest in F[1..n]
3. Remove j and all jobs that conflict with j from S,F.
4. Return $1 +$ Greedy(S,F after removals)

}

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If n $=$ 1 return 1
2. j $=$ integer s.t. F[j] smallest in F[1..n]
3. Remove j and all jobs that conflict with j from S,F.
4. Return 1 $+$ Greedy(S,F after removals)

}

Running time: $T(n) =$

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. j = integer s.t. F[j] smallest in F[1..n]
3. Remove j and all jobs that conflict with j from S,F.
4. Return 1 + Greedy(S,F after removals)

}

Running time: $T(n) = O(n) + T(n-1)$

Algorithm

For convenience we will determine the number of jobs in the optimal solution.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. j = integer s.t. F[j] smallest in F[1..n]
3. Remove j and all jobs that conflict with j from S,F.
4. Return 1 + Greedy(S,F after removals)

}

Running time: $T(n) = O(n) + T(n-1) = O(n^2)$

Improvement:

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. k = smallest integer s.t. S[j] > F[1]

Job 1 conflicts with Jobs 2..j-1

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. k = smallest integer s.t. S[j] > F[1]

   Job 1 conflicts with Jobs 2..j-1

3. Return 1 + Greedy(S[j..n],F[j..n])

}

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. k = smallest integer s.t. S[j] > F[1]

   Job 1 conflicts with Jobs 2..j-1

3. Return 1 + Greedy(S[j..n],F[j..n])

}

$T(n) = O(k) + T(n-k) = O(n)$

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. k = smallest integer s.t. S[j] > F[1]

   Job 1 conflicts with Jobs 2..j-1

3. Return 1 + Greedy(S[j..n],F[j..n])

}

$T(n) = O(k) + T(n-k) = O(n)$ Total time $= O(n \log n)$

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. k = smallest integer s.t. S[j] > F[1]

   Job 1 conflicts with Jobs 2..j-1

3. Return 1 + Greedy(S[j..n],F[j..n])

}

$T(n) = O(k) + T(n-k) = O(n)$ Total time $= O(n \log n)$

Subtle point: Some job $k > j$ may conflict with job 1. But it will also conflict with job j.

Improvement:

Preprocess: Sort S, F by F. Time $= O(n \log n)$.

Greedy(S[1..n],F[1..n]){

1. If n = 1 return 1
2. k = smallest integer s.t. S[j] > F[1]

   Job 1 conflicts with Jobs 2..j-1
3. Return 1 + Greedy(S[j..n],F[j..n])

}

$T(n) = O(k) + T(n-k) = O(n)$ Total time $= O(n \log n)$

Subtle point: Some job $k > j$ may conflict with job 1. But it will also conflict with job j.
Job j will be selected in the next recursion and so k will not be selected.

Remarks

Remarks

- In problems involving intervals, it is often difficult to decide what is the right greedy strategy. Try all possible orders, e.g. earliest/latest start/finish, and also biggest interval, interval with least conflicts etc. Hope something works!

Remarks

- In problems involving intervals, it is often difficult to decide what is the right greedy strategy. Try all possible orders, e.g. earliest/latest start/finish, and also biggest interval, interval with least conflicts etc. Hope something works!
- Earliest finishing job may not be in every optimal solution. We only proved that there exists an optimal solution which contains an earliest finishing job.

Remarks

- In problems involving intervals, it is often difficult to decide what is the right greedy strategy. Try all possible orders, e.g. earliest/latest start/finish, and also biggest interval, interval with least conflicts etc. Hope something works!
- Earliest finishing job may not be in every optimal solution. We only proved that there exists an optimal solution which contains an earliest finishing job.
- The rest of the solution is found by doing more work: luckily this work turns out to be an instance of the same kind, so recursion works.

Remarks

- In problems involving intervals, it is often difficult to decide what is the right greedy strategy. Try all possible orders, e.g. earliest/latest start/finish, and also biggest interval, interval with least conflicts etc. Hope something works!
- Earliest finishing job may not be in every optimal solution. We only proved that there exists an optimal solution which contains an earliest finishing job.
- The rest of the solution is found by doing more work: luckily this work turns out to be an instance of the same kind, so recursion works.
- As always, some sorting helps reduce the time from $O(n^2)$ to $O(n \log n)$.

How to design greedy algorithms

How to design greedy algorithms

- Propose a way to take the first decision: "Imagine the optimal solution, make a small change, ..."

How to design greedy algorithms

- Propose a way to take the first decision: "Imagine the optimal solution, make a small change, ..." Greedy choice

How to design greedy algorithms

- Propose a way to take the first decision: "Imagine the optimal solution, make a small change, ..." Greedy choice
- Prove a greedy choice theorem: "There exists an optimal solution in which the first decision can be made in the proposed manner."

How to design greedy algorithms

- Propose a way to take the first decision: "Imagine the optimal solution, make a small change, ..." Greedy choice
- Prove a greedy choice theorem: "There exists an optimal solution in which the first decision can be made in the proposed manner."
- Show that the task of taking the remaining decisions = solving another instance of the same type. (residual instance)

How to design greedy algorithms

- Propose a way to take the first decision: "Imagine the optimal solution, make a small change, ..." Greedy choice
- Prove a greedy choice theorem: "There exists an optimal solution in which the first decision can be made in the proposed manner."
- Show that the task of taking the remaining decisions = solving another instance of the same type. (residual instance)
- Prove an Optimal substructure theorem: "Given an optimal solution to the residual instance, we can use that to build a solution to the original instance."

How to design greedy algorithms

- Propose a way to take the first decision: "Imagine the optimal solution, make a small change, ..." Greedy choice
- Prove a greedy choice theorem: "There exists an optimal solution in which the first decision can be made in the proposed manner."
- Show that the task of taking the remaining decisions = solving another instance of the same type. (residual instance)
- Prove an Optimal substructure theorem: "Given an optimal solution to the residual instance, we can use that to build a solution to the original instance."
- Other ways also possible, soon.

Interval graph colouring

Interval graph colouring

Input: S[1..n], F[1..n]

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]

Output: Minimum number of rooms needed to accommodate all guests.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R - 1$ rooms were occupied, and a guest was waiting.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R-1$ rooms were occupied, and a guest was waiting.
Thus R rooms are necessary.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R-1$ rooms were occupied, and a guest was waiting.
Thus R rooms are necessary. $\square$

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R-1$ rooms were occupied, and a guest was waiting.
Thus R rooms are necessary. $\square$

Algorithm: Replace S[i] by triple (s,i,S[i]), and R[i] by (f,i,F[i]).

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R-1$ rooms were occupied, and a guest was waiting.
Thus R rooms are necessary. □

Algorithm: Replace S[i] by triple (s,i,S[i]), and R[i] by (f,i,F[i]).
Sort the triples by 3rd value (time).

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R-1$ rooms were occupied, and a guest was waiting.
Thus R rooms are necessary. □

Algorithm: Replace S[i] by triple (s,i,S[i]), and R[i] by (f,i,F[i]).
Sort the triples by 3rd value (time).
Process the triples in time order, keeping track of guests as they arrive and leave, placing them in available or new rooms.

Interval graph colouring

Input: S[1..n], F[1..n]   Guest i arrives on date S[i], leaves on F[i]
Output: Minimum number of rooms needed to accommodate all guests.

"Lazy" algorithm: Serve guests in arrival order. If there is an empty room (vacated by some guest), give it. If there is no empty room, build a room.

Claim: Total number of rooms built = minimum number needed.
Proof: Suppose you build R rooms.
At the time of building Rth room, $R-1$ rooms were occupied, and a guest was waiting.
Thus R rooms are necessary. □

Algorithm: Replace S[i] by triple (s,i,S[i]), and R[i] by (f,i,F[i]).
Sort the triples by 3rd value (time).
Process the triples in time order, keeping track of guests as they arrive and leave, placing them in available or new rooms.
Time $= O(n \log n)$ for sorting.

Remarks

Remarks

- Assigned room = "colour". Each guest is being coloured so that conflicting guests get different colours.

Remarks

- Assigned room = "colour". Each guest is being coloured so that conflicting guests get different colours.
- Optimality proof is very simple. We do not bother to compare with the optimal algorithm: we directly show that the number of rooms cannot be too small.