More about NPC Problems

Algorithm : Design & Analysis

[21]

In the Last Class…

- Decision Problem
- The Class *P*
- The Class *NP*
- *NP*-Complete Problems
  - Polynomial Reductions
  - *NP*-hard and *NP*-complete

More about NPC Problems

- Polynomial Reduction
- Conquer the Complexity by Approximation
- Approximation Algorithm for Bin Packing
- Evaluate an Approximation Algorithm
- Online algorithm

NPC as a Level of Complexity

- If *P* is a *NP*-complete problem, then:
  - If there is a polynomial bounded algorithm for *P*, then there would be a polynomial bounded algorithm for **each** of the problems in *NP*.
  - *P* is as **difficult** to be solved as that no problem in *NP* is more difficult than *P*; and *P* is as **easy** to be solved as that there exists a polynomially bounded nondeterministic algorithm which solves *P*.

Satisfiability Problem

- CNF
  - A literal is a Boolean variable or a negated Boolean variable, as x or $\overline{x}$
  - A clause is several literals connected with $\vee$s, as $(x_1 \vee \overline{x}_2)$
  - A CNF formula is several clause connected with $\wedge$s
- CNF-SAT problem
  - Is a given CNF formula satisfiable, i.e. taking the value TRUE on some assignments for all x_i.
- A special case: 3-CNF-SAT

Proving *NPC* by Reduction

- The *CNF-SAT* problem is *NP*-complete. (so is 3-*CNF-SAT*)
- Prove problem Q is *NP*-complete, given a problem P known to be *NP*-complete
  - For all $R \in NP$, $R \le_P P$;
  - **Show $P \le_P Q$;**
  - By transitivity of reduction, for all $R \in NP$, $R \le_P Q$;
  - So, Q is *NP*-hard;
  - If Q is in *NP* as well, then Q is *NP*-complete.

Max Clique Problem is in $\mathcal{NP}$

void nondeteClique(graph G; **int** n, k)
  set S=ϕ;
  for int i=1 **to** k **do**
    int t=genCertif();
    if $t \in S$ **then return**;
    $S=S\cup\{t\}$;
  for all pairs (i,j) with i,j in S and $i \neq j$ **do**
    if (i,j) is not an edge of G
      then return;
    Output("yes");

$\rightarrow$ In $\boldsymbol{O(n)}$

$\rightarrow$ In $\boldsymbol{O(k^2)}$

So, we have an algorithm for the maximal clique problem with the complexity of $O(n+k^2)=\boldsymbol{O(n^2)}$

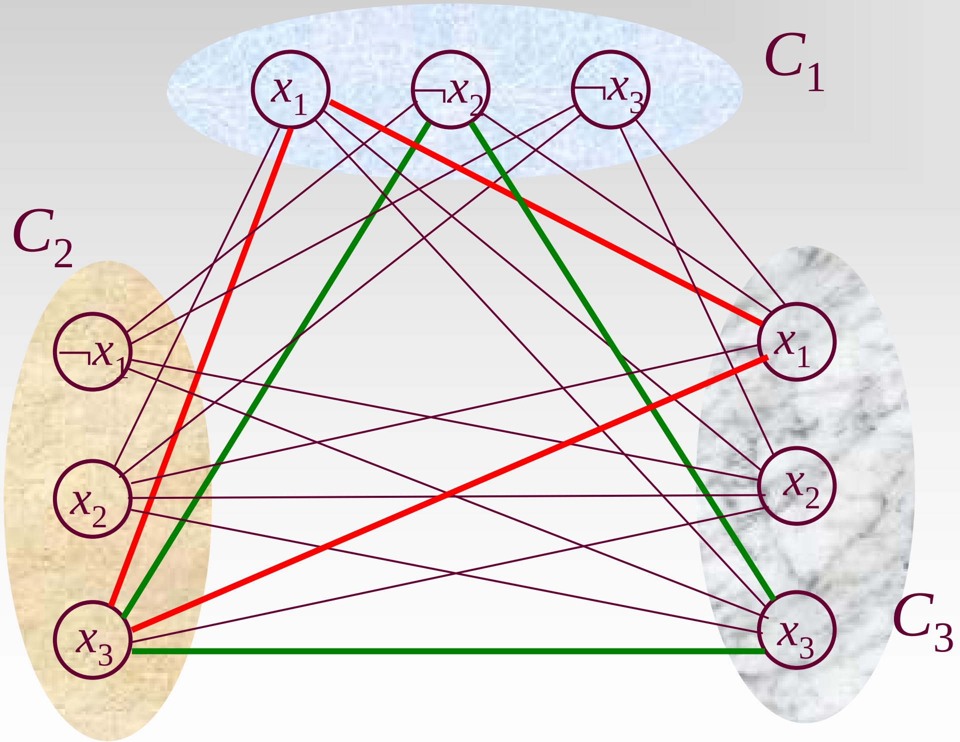
CNF-SAT to Clique

- Let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a formula in 3-*CNF* with k clauses. For $r = 1,2,\ldots,k$, each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$, l_i^r is x_i or $\neg x_i$, any of the variables in the formula.
- A graph can be constructed as follows. For each C_r, create a triple of vertices v_1^r, v_2^r and v_3^r, and create edges between v_i^r and v_j^s if and only if:
  - they are in different triples, i.e. $r \neq s$, and
  - they do not correspond to the literals negating each other

(Note: there is no edges within one triple)

The Graph Corresponding 3-CNF

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



Two of satisfying assignments:

x_1=1/0, x_2=0; x_3=1, or

x_1=1, x_2=1/0, x_3=1

For corresponding clique, pick one "true" literal from each triple

Clique Problem is $\mathcal{NP}$-Complete

- ϕ, with k clauses, is satisfiable if and only if the corresponding graph G has a clique of size k.
- Proof: $\Rightarrow$
  - Suppose that ϕ has a satisfying assignment.
  - Then **there is at least one "true" literal in each clause**. Picking such a literal from each clause, their corresponding vertices in G can be proved to be a clique, since any two of them are in different triples and cannot be complements to each other(they are both true).

Clique Problem is $\mathcal{NP}$-Complete

- ϕ, with k clauses, is satisfiable if and only if the corresponding graph G has a clique of size k.
- Proof: $\Leftarrow$
  - Suppose that G has a clique V' of size k.
  - Note there is no edge within one triple, so V' contains exactly one vertex from each triple. Assigning "true" to the literal corresponding to every vertices in V', no inconsistency will be resulted according to the rule by which the graph is constructed. The assignment is a satisfying assignment. (The variables whose corresponding vertices are not in V' can be assigned either 0 or 1.)

Conquer the Complexity

- Challenge: more than often, the problem with important practical background is NPC.
- Good algorithm in “general sense: functionally perfection and efficiency
- Not perfect, but efficiency: approximation
- Low probability input ignored: probability

Bin Packing Problem

- Suppose we have an unlimited number of bins each of capacity one, and *n* objects with sizes $s_1, s_2, \ldots, s_n$ where $0 < s_i \leq 1$ (s_i are rational numbers)
- ***Optimization problem***: Determine the smallest number of bins into which the objects can be packed (and find an optimal packing) .
- Bin packing is a NPC problem

Feasible Solution

- For any given input $I=\{s_1,s_2,\ldots,s_n\}$, the feasible solution set, $FS(I)$ is the set of all **valid packings** using any number of bins.
- In other word, that is the set of all partitions of I into disjoint subsets $T_1,T_2,\ldots,T_p$, for some p, such that the **total of the s_i in any subset is at most 1**.

Optimal Solution

- In the bin packing problem, the **optimization parameter** is the number of bins used.
- For any given input I and a feasible solution x, ***val***(I,x) is the value of the optimization parameter.
- For a given input I, the optimum value, ***opt***$(I) = \min\{val(I,x) \mid x \in FS(I)\}$
- An optimal solution for I is a feasible solution which achieves the optimum value.

Approximation Algorithm

- An **approximation algorithm** for a problem is a polynomial-time algorithm that, when given input I, output an element of $FS(I)$.
- Quality of an approximation algorithm.

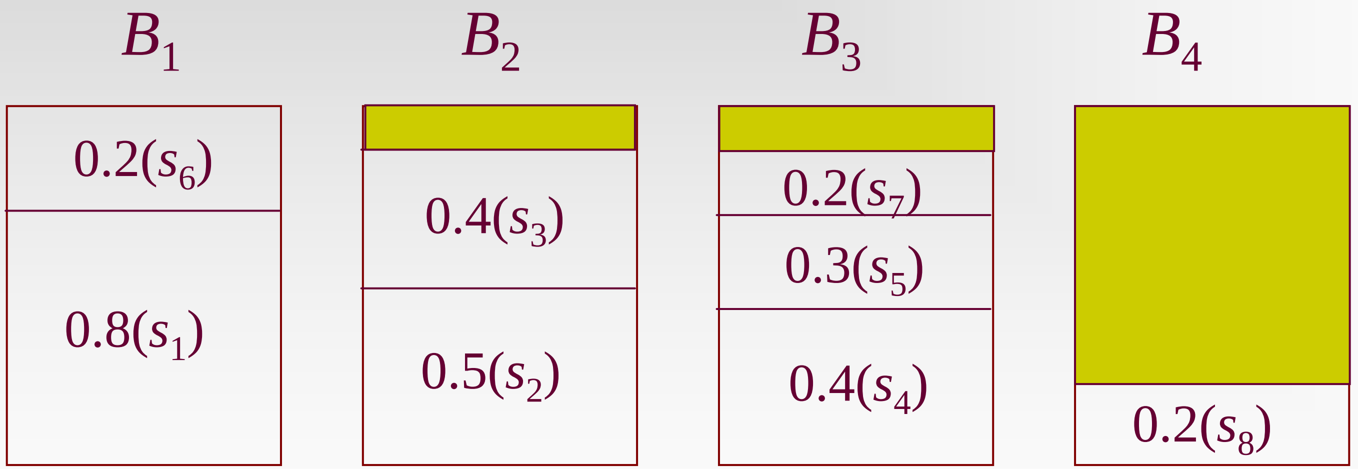
$$r_{\mathrm{A}}(I) = \frac{val(I, \mathrm{A}(I))}{opt(I)}$$

$$R_{\mathrm{A}}(m) = \max \{ r_{\mathrm{A}}(I) \mid I \text{ such that } opt(I) = m \}$$

- For an approximation algorithm, we hope the value of $R_{\mathrm{A}}(m)$ is bounded by small constants.

First Fit Decreasing - FFD

- The strategy: packing the largest as possible
- Example: S=(0.8, 0.5, 0.4, 0.4, 0.3, 0.2, 0.2, 0.2)



This is ***NOT*** an optimal solution!

The Procedure

```
binpackFFD(S, n, bin) //bin is filled and output, object i is packed in bin[i]
    float[] used=new float[n+1]; //used[j] is the occupied space in bin j
    int i,j;
    <initialize all used entries to 0.0>
    <sort S into nonincreasing order> // in S after sorted
    for (i=1; i≤n; i++)
        for (j=1; j≤n; j++)
            if (used[j]+S[i]≤1.0)
                bin[i]=j;
                used[j]+=S[i];
                break;
```

in $O(n\log n)$

i, at most

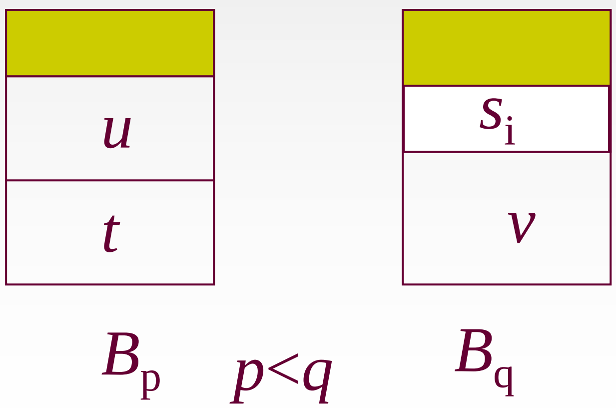
so, $n^2/2$

Small Objects in Extra Bins

- Let $S=\{s_1, s_2, \ldots, s_n\}$ be an input, **in non-increasing order**, for the bin packing problem and let $opt(S)$ be the minimum number of bins for S. All of the objects placed by FFD in the extra bins have size at most 1/3.
- Let i be the index of the first object placed by FFD in bin $opt(S)+1$. What we have to do for the proof is: $s_i \leq 1/3$.

What about a s_i larger than 1/3?

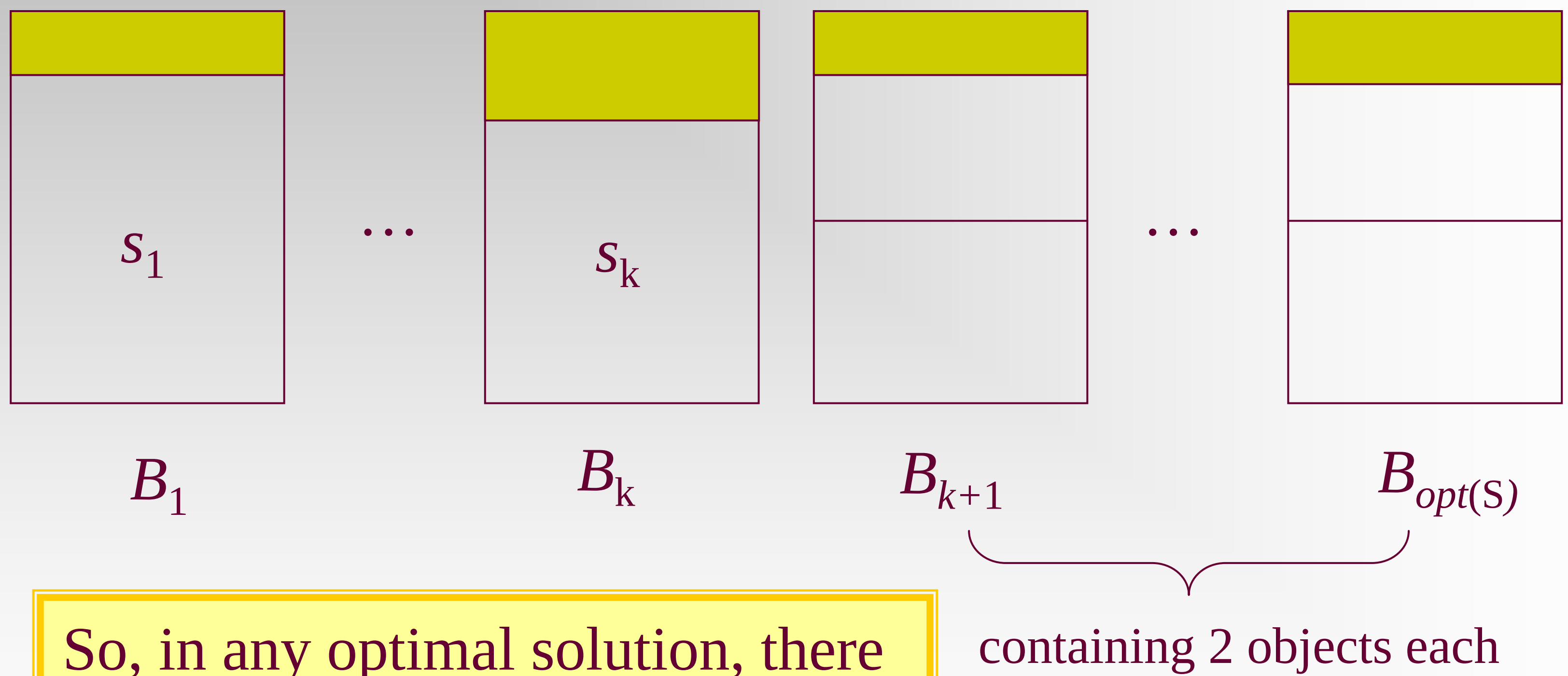
- [S is sorted] The $s_1, s_2, \ldots, s_{i-1}$ are all larger than 1/3.
- So, bin B_j for $j=1,\ldots,opt(S)$ contain at most 2 objects each.
- Then, for some $k \geq 0$, the first k bins contain one object each and the remaining $opt(S)-k$ bins contain two each.
  - Proof: no situation (that is, some bin containing 2 objects has a smaller index than some bin containing only one object) as the following is possible



Then: we must have:

$t>v$, $u>s_i$, so $v+s_i<1$, no extra bin is needed!

View when Considering s_i



So, in any optimal solution, there will be *k* bins that do not contain any of the objects k+1,…, i.

Contradicting at Last!

- Any optimal solution use only *opt*(*S*) bins.
- However, there are k bins that do not contain any of the objects k+1, …, i-1, i. k+1,…, i-1 must occupy *opt*(*S*)-k bins, with each bin containing 2.
- Since all objects down through to s_i are larger than 1/3, s_i can not fit in any of the *opt*(*S*)-k bins.
- So, extra bin needed, and contradiction.

Objects in Extra Bins is Bounded

- For any input $S=\{s_1, s_2,\ldots,s_n\}$, the number of objects placed by FFD in extra bins is at most $opt(S)$-1.

Since all the objects fit in $opt(S)$, $\sum_{i=1}^{n} s_i \le opt(S)$.

Assuming that FFD puts $opt(S)$ objects in extra bins,

and their sizes are: $t_1, t_2, \ldots, t_{opt(S)}$.

Let b_j be the final contents of bin B_j for $1 \le j \le opt(S)$.

Note $b_j + t_j > 1$, otherwise t_j should be put in B_j. So:

$$\sum_{i=1}^{n} s_i \ge \sum_{j=1}^{opt(S)} b_j + \sum_{j=1}^{opt(S)} t_j = \sum_{j=1}^{opt(S)} (b_i + t_i) > opt(S); \text{Contradiction!}$$

A Good Approximation

- ***Using FFD, the number of bin used is at most about 1/3 more than optimal value.***

$$R_{FFD}(m) \leq \frac{4}{3} + \frac{1}{3m}$$

FFD puts at most *m*-1 objects in extra bins, and the size of the *m*-1 object are at most 1/3 each, so, FFD uses at most $\lceil (m\text{-}1)/3 \rceil$ extra bins.

$$r_{FFD}(S) \leq \frac{m + \left\lceil \frac{m-1}{3} \right\rceil}{m} \leq 1 + \frac{m+1}{3m} \leq \frac{4}{3} + \frac{1}{3m}$$

Average Performance Is Much Better

- Empirical Studies on large inputs.
- The number of extra bins are estimated by the amount of empty space in the packings produced by the algorithm.
- It has been shown that for n objects with sizes uniformly distributed between zero and one, the expected amount of empty space in packings by FFD is approximately $0.3 \sqrt{n}$.

Online Algorithm

- The entries are input one by one, which means that the algorithm never know when the input will end before it ends.
- For the bin packing, each item must be placed in a bin before the next item can be processed.
- Since the input may put up the end at any time, so, any optimal solution should be optimal at any time on the processed input.

Challenge for Online Algorithm

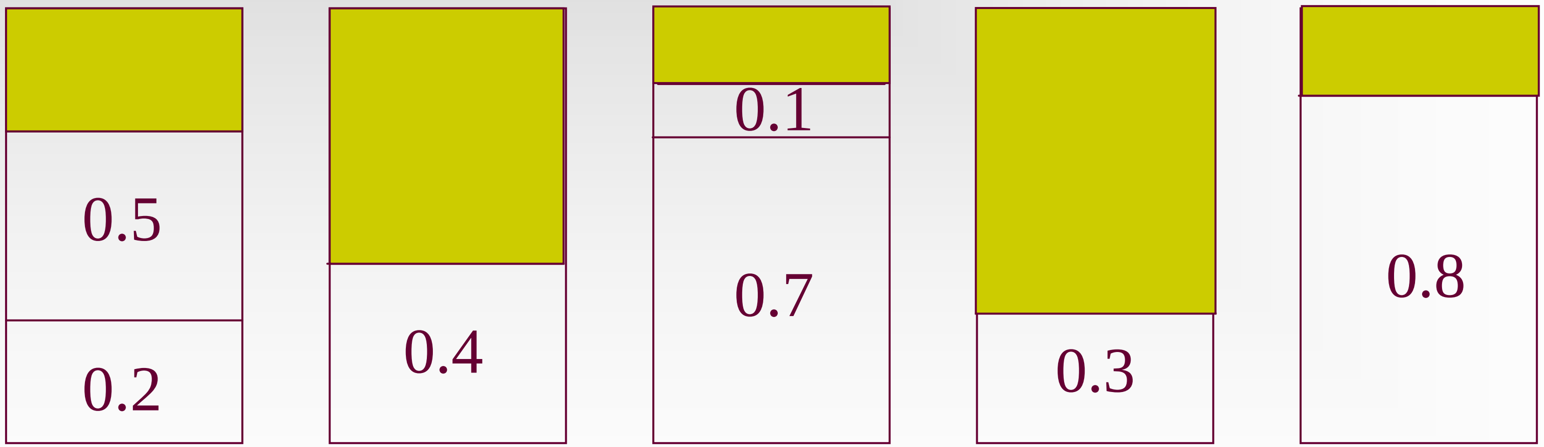
- Even though unlimited computation is allowed, an online algorithm cannot always give an optimal solution.
  - Input 1: n object with the value 0.5- ε each, following by n object with the value 0.5+ε each; ($0<\varepsilon<0.01$)
  - Input 2: only n object with the value 0.5-ε each
  - No algorithm can give optimal solution for both inputs

Lower Bound for Online Algorithms

- Any on-line bin-packing algorithm use **at least 4/3** the optimal number of bins for the worst case.
- Proof
  - Assume that the performance guarantee is better than 4/3. Consider the input S of n items of size ½-ε(0<ε<0.01) followed by n items of size ½+ε. After the nth item is processed, the algorithm uses b bins, and, at the time, the optimal number of bins is $n/2$. So, $2b/n<4/3$, i.e. $\boldsymbol{b/n<2/3}$;
  - However, after all $2n$ items are processed, the algorithm uses at least $2n-b$ bins (at most n bins contain 2 items each). So, $(2n-b)/n<4/3$, which means $\boldsymbol{b/n>2/3}$, contradiction.

Next Fit Algorithm - NF

- The strategy: Put a new item in the last bin if possible, or use a new bin. Never look back!
- An example: $S=\{0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8\}$



Simple, but Not So Bad

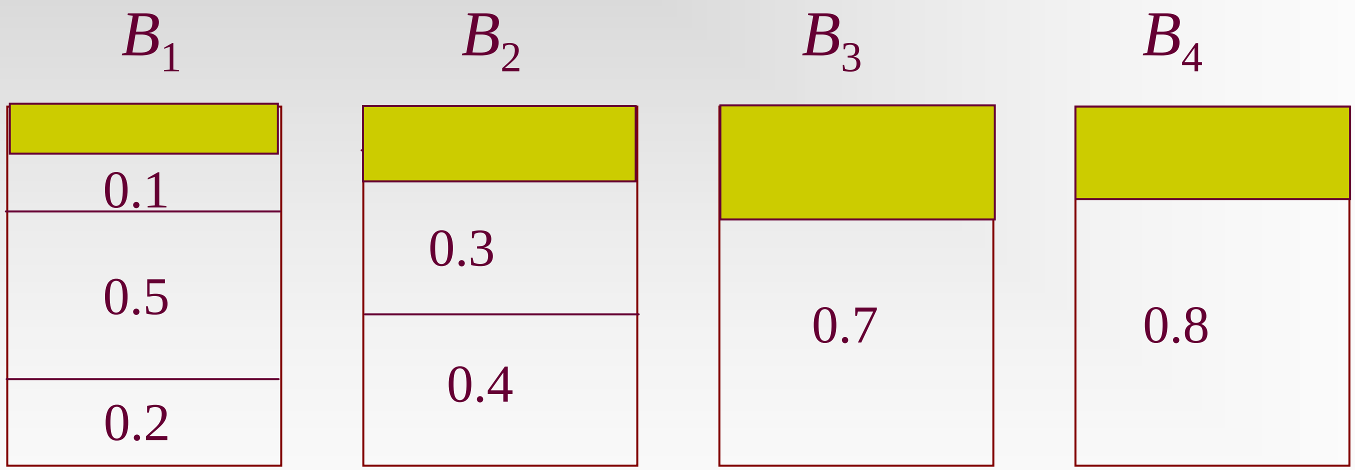
- For a given *S*, if the optimal value is *m*, NF algorithm uses at most 2*m* bins.
- Proof:
  - Note that no two consecutive bins can contain object with total value less than 1, which mean half space is wasted at most.

A Tight Bound for NF

- There exists input for which NF uses $2m$-2 bins.
- An example: The input sequence S consists of n items (n=4k for some integer k). The size of s_i for odd i is 0.5 and size of s_i for even i is 2/n.
- The optimal number of bins is n/4+1(n/4 for 2 of 0.5 each, and 1 for all items with the size 2/n)
- NF uses n/2 bins.

First Fit Algorithm - FF

- Creating new bin only when necessary, the algorithm scans all used bins to look for one able to hold a new item.
- For the same example: S={0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8}



The FF never uses more than $\lceil 1.7m \rceil$ bins

Home Assignments

- pp. 600-
  - 13.10
  - 13.14
  - 13.17
  - 13.29
  - 13.31-33