```
#include⟨stdio.h⟩
#include"schedule.h"

int main(void) {
  int i, z = 0; struct node root = { 0, 0, 0} ;
  create_jobs(); insert(root);
  while(!isempty()) {
    struct node E = extract(), E1, E2; int k = E.k;
    z++;
    if(k == N) {
      for(i = 0; i < N; i++) printf("%d ", E.x[i]);
      printf("-> %d\n", E.p);
      continue;
    }
    E1 = E; E1.k = k + 1; E1.x[k] = 0; E1.p += jobs[k].p;
    E2 = E; E2.k = k + 1; E2.t += jobs[k].t; E2.x[k] = 1;
    insert(E1);
    if(E2.t ≤ jobs[k].d) insert(E2);
  }
  printf("Nodes: %d\n", z);
  return 0;
}
```

Implementierung für LIFO-Variante:

```
#include"schedule.h"

int l = 0;
struct node life[200000];

void insert(struct node n)
{
  if(l == 200000) exit(100);
  life[l++] = n;
}

struct node extract(void)
{
  return life[--l];
}

int isempty(void)
{
  return l == 0;
}
```

Ausgabe für LIFO:

```
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 -> 3717
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 0 -> 3728
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 -> 3670
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 0 -> 3681
.
.
.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 -> 8966
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 -> 9348
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 -> 9359
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 -> 9024
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 -> 9035
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 -> 9417
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -> 9428
Nodes: 459009
```

```
#include⟨stdio.h⟩
#include"schedule.h"
int main(void) {
  int i, z = 0, upper = 100000; struct node root = { 0, 0, 0} ;
  create_jobs(); insert(root);
  while(!isempty()) {
    struct node E = extract(), E1, E2;
    int k = E.k, u = E.p; z++;
    for(i = k; i < N; i++) u += jobs[i].p;
    if(u < upper) upper = u;
    if(k == N) {
      for(i = 0; i < k; i++) printf("%d ", E.x[i]);
      printf("-> %d\n", E.p);
      continue;
    }
    E1 = E; E1.k = k + 1; E1.x[k] = 0; E1.p += jobs[k].p;
    E2 = E; E2.k = k + 1; E2.t += jobs[k].t; E2.x[k] = 1;
    if(E1.p ≤ upper) insert(E1);
    if(E2.p ≤ upper && E2.t ≤ jobs[k].d) insert(E2);
  }
  printf("Nodes: %d\n", z); return 0;
}
```

Ausgabe für FIFO Branch-and-Bound:

```
0 1 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2914
0 1 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2903
0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 0 1 1 0 -> 2886
0 1 0 0 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 -> 2877
0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 0 1 0 -> 2783
0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 -> 2772
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2593
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2582
Nodes: 9726
```

Ausgabe für LIFO Branch-and-Bound:

```
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 -> 3717
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 0 -> 3728
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 -> 3670
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 0 -> 3681
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 1 -> 3346
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 0 -> 3357
0 1 0 1 1 1 0 1 0 1 0 0 1 1 0 1 0 0 1 1 -> 3342
0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 -> 3303
0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 -> 3314
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 -> 3271
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 0 -> 3282
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 0 1 1 -> 2947
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 0 1 0 -> 2958
0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1 0 0 1 1 -> 2842
0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1 0 0 1 0 -> 2853
0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 -> 2769
0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 0 -> 2780
0 1 0 0 1 1 1 0 1 0 1 1 1 1 0 1 0 0 1 1 -> 2655
0 1 0 0 1 1 1 0 1 0 1 1 1 1 0 1 0 0 1 0 -> 2666
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2582
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2593
Nodes: 348
```

Ausgabe für LC Branch-and-Bound:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2582
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2593
Nodes: 303
```

Hier wurden bereits im Inneren des Suchbaums so gute Schranken berechnet, daß im Endeffekt nur zwei Blätter besucht werden mußten.

Selbst für $N = 40$ bleibt der Suchbaum angenehm klein:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 -> 3540
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 -> 3583
Nodes: 19969
```

Eine bessere Schranke

Der LC-Suchalgorithmus berechnete eine Schranke für die zu zahlende Strafe, indem er die Strafen der noch nicht betrachteten Aufgaben zur bisher zu zahlenden Strafe addierte.

Dies ist eine sehr konservative Abschätzung.

Wir können diese Abschätzung leicht Verbessern:

Seien $x_1, \ldots, x_{k-1}$ schon festgelegt. Wir gehen $i = k, \ldots, n$ durch und dabei setzen $x_i = 1$, falls dadurch die Deadline für die Aufgabe i nicht verletzt wird.

Dies führt zu einer Menge von Aufgabe, die ohne Strafe durchgeführt werden können.

```
#include⟨stdio.h⟩
#include"schedule.h"
int main(void)
{
  int i, z = 0, upper = 100000; struct node root = { 0, 0, 0} ;
  create_jobs(); insert(root);
  while(!isempty()) {
    struct node E = extract(), E1, E2;
    int k = E.k, u = E.p, ut = E.t; z++;
    for(i = k; i < N; i++)
      if(ut + jobs[i].t ≤ jobs[i].d) ut += jobs[i].t;
      else u += jobs[i].p;
    if(u < upper) upper = u;
    if(k == N) {
      for(i = 0; i < k; i++) printf("%d ", E.x[i]);
      printf("-> %d\n", E.p);
      continue; }
    E1 = E; E1.k = k + 1; E1.x[k] = 0; E1.p += jobs[k].p;
    E2 = E; E2.k = k + 1; E2.t += jobs[k].t; E2.x[k] = 1;
    if(E1.p ≤ upper) insert(E1);
    if(E2.p ≤ upper && E2.t ≤ jobs[k].d) insert(E2);
  }
```

```
  printf("Nodes: %d\n", z); return 0;
}
```

$N = 40$, LC-Suche ohne Heuristik:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 -> 3540
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 -> 3583
Nodes: 19969
```

$N = 40$, LC-Suche mit Heuristik:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 -> 3540
Nodes: 12867
```

Mit Heuristik: 5 mal weniger Speicherverbrauch

(Bei LIFO allerdings noch viel, viel, viel weniger.)

Approximationsalgorithmen

In polynomieller Zeit lässen sich nicht exakte Lösungen von *NP*-harten Problemen berechnen.

Approximationsalgorithmen versuchen das beste aus einer Polynomialzeitberechnung zu machen, indem sie wenigstens eine Näherunslösung bestimmen.

Im Gegensatz zu Heuristiken, verlangen wir aber von Approximationsalgorithmen eine Garantie, daß die berechnete Näherungslösung nah am Optimum ist.

Approximationsalgorithmen

Sei A ein Algorithmus, der in polynomieller Zeit eine Näherungslösung zu einem gegebenen Optimierungsproblem berechnet.

Sei $F^*(I)$ der optimale Wert der Zielfunktion und $F(I)$ der vom Algorithmus A erzielte Wert.

Definition

Das *Approximationsverhältnis* von A ist α, wenn

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \alpha \text{ und } \frac{F^*(I) - F(I)}{F^*(I)} \leq \alpha$$

gilt.

Approximationsalgorithmen

Diese Definition ist sowohl für Minimierungs- als auch für Maximierungsprobleme sinnvoll.

Für Minimierungsprobleme ist diese Ungleichung ausschlaggebend:

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \alpha$$

Für Maximierungsprobleme dagegen die andere:

$$\frac{F^*(I) - F(I)}{F^*(I)} \leq \alpha$$

Beispiel: Vertex Cover

Der Approximationsfaktor ist 2.

$C := \emptyset$;
while $E \neq \emptyset$ **do**
  choose some $e \in E$;
  $V := V - e$;
  $C := C \cup e$;
  $E := \{ e' \in E \mid e \cap e' = \emptyset \}$
od;
return C

Kompetitive Analyse

Um das Approximationsverhältnis zu analysieren, bietet sich eine *kompetitive Analyse* der folgenden Form an:

Wir gehen von einer Instanz und einer zugehörigen optimalen Lösung aus.

Wir müssen dann beweisen, daß der Approximationsalgorithmus eine Lösung findet, die nur um α schlechter als die vorgegebene optimale Lösung ist.

Kompetitive Analyse

Im Falle von Vertex Cover ist eine kompetitive Analyse recht einfach:

Sei $G = (V, E)$ ein Graph und $C \subseteq V$ ein minimales Vertex Cover.

Der Algorithmus wählt eine Kante $e = \{v_1, v_2\} \in E$ aus fügt v_1 und v_2 zu seinem VC hinzu. Einer von beiden Knoten ist aber auch in C. Zu einem Knoten in C, fügt der Algorithmus also höchstens 2 zu seiner Lösung hinzu.

Das macht ein Verhältnis von höchstens 2 zwischen konstruierter und optimaler Lösung aus.

Approximationsalgorithmen

Definition

1. Ein Approximationsalgorithmus ist ein *$f(n)$-Approximationsalgorithmus*, wenn das Approximationsverhältnis höchstens $f(n)$ für alle Instanzen der Länge n ist.
2. Ein *ϵ-Approximationsalgorithmus* ist ein $f(n)$-Approximationsalgorithmus mit $f(n) \leq \epsilon$ für alle $n \in \mathbf{N}$.